FREE!
OVER 4 HOURS OF EXCLUSIVE TUTORIAL VIDEOS

From the makers of
net

THE JAVASCRIPT HANDBOOK

The ultimate guide to enhancing your sites with the most important programming language in the world

27 PRACTICAL PROJECTS

DISCOVER HOW TO:

- BUILD APPS WITH JS FRAMEWORKS
- SPEED UP YOUR SITES
- GET THE MOST OUT OF GRUNT
- GO 3D WITH WEBGL
- PREPARE FOR THE INTERNET OF THINGS
- CREATE STUNNING USER INTERFACES

228 PAGES OF PRO ADVICE

Future

The JavaScript Handbook

INTRODUCTION

THE JAVASCRIPT HANDBOOK

The internet as we know it wouldn't really work without JavaScript. At one point JavaScript was considered just a toy language, a language for browsers. But just look at it now: it powers node.js, three.js, jQuery and many of the most powerful frameworks out there – from AngularJS to Ampersand.js. The list of JavaScript libraries is seemingly endless.

In this new guide from the makers of **net** magazine, we present the essential JavaScript tools and technologies you need to build better sites and apps. You'll discover how to create web, mobile and single-page apps with JavaScript frameworks, how to speed up your sites' performance, how to create stunning user interfaces and how to master animated 3D graphics with WebGL.

But that's not all. JavaScript is now being used to control hardware: your mobile phone, the thermostat in your house, robotics …

Over the next 228 pages we'll look at the present and future of JavaScript. There are 27 practical projects (and more than four hours of video tutorials) to help you get your hands dirty and dig right into the JavaScript. Happy coding!

Oliver Lindberg, editor
oliver.lindberg@futurenet.com
@oliverlindberg

FEATURED AUTHORS

JOHN ALLSOPP

Author and developer John is the self-proclaimed 'great grandfather' of RWD. On page 56 he welcomes in the sensory web
w: *johnfallsopp.com*
t: @johnallsopp

JACK FRANKLIN

Jack works at GoCardless, writes at *javascriptplayground.com*, and from page 6 lists all the top JavaScript tools you can't afford to miss
w: *jackfranklin.co.uk*
t: @Jack_Franklin

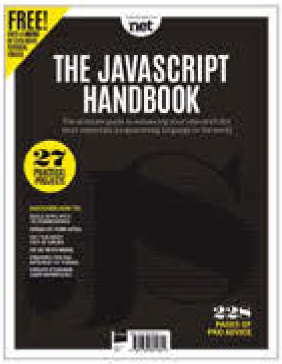
PATRICK H LAUKE

Patrick is an accessibility consultant at The Paciello Group. On page 166 he explains how to make your sites work on touch devices
w: *splintered.co.uk*
t: @patrick_h_lauke

NICHOLAS ZAKAS

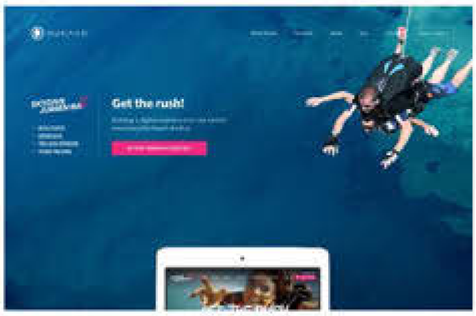
Nicholas is a frontend engineer, author and speaker, currently working at Box. On page 118 he presents the four JavaScript loading times
w: *nczonline.net*
t: @slicknet

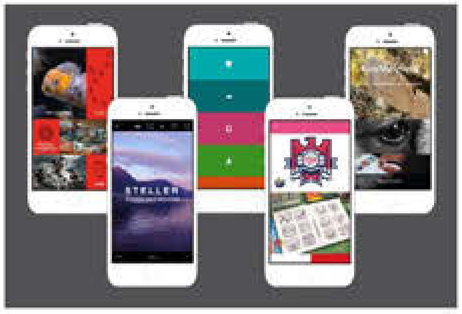
CONTENTS


FREE!
net
THE JAVASCRIPT HANDBOOK
27 PRACTICAL PROJECTS
228 PAGES OF PRO ADVICE

node
HANDLEBARS
JS
ember.js
Gulp
BOWER
YEOMAN
ANGULAR

OPTIMIZE for CHANGE
it's the only CONSTANT


Get the rush!


STELLER

p5.js

THE
Complete
WEB DESIGN
TOOLKIT
VIM
Git
COFFEE SCRIPT
Sublime
Sass
ASSEMBLE
GRUNT

Contents

TOOLS & TECHNOLOGIES

GALLERY

THE FUTURE OF JAVASCRIPT

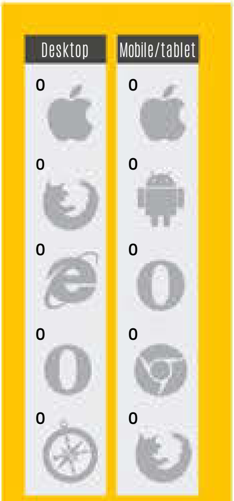
BUILD APPS WITH FRAMEWORKS

PERFORMANCE & WORKFLOW

UI & RWD

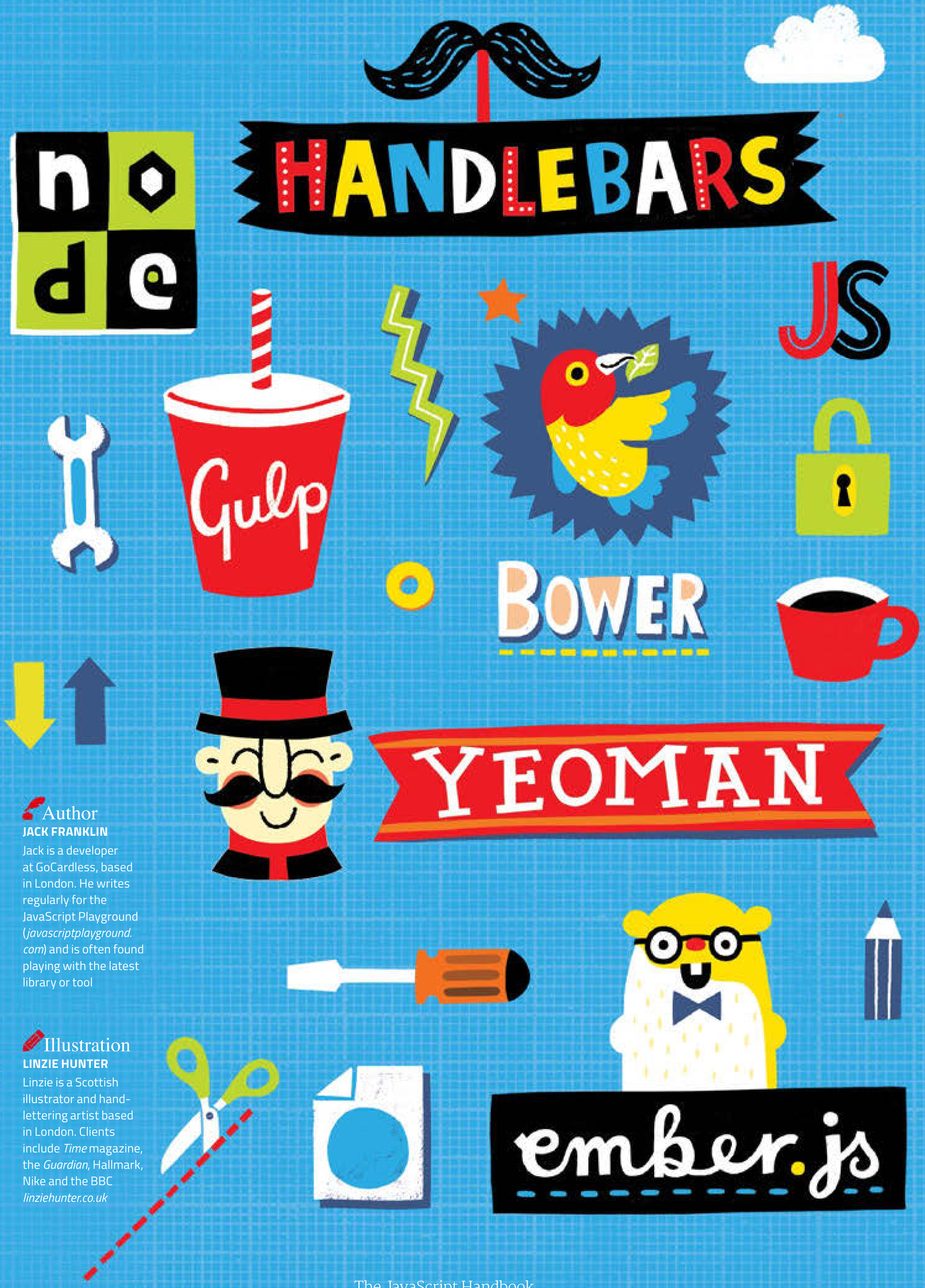
WEBGL

EXCHANGE



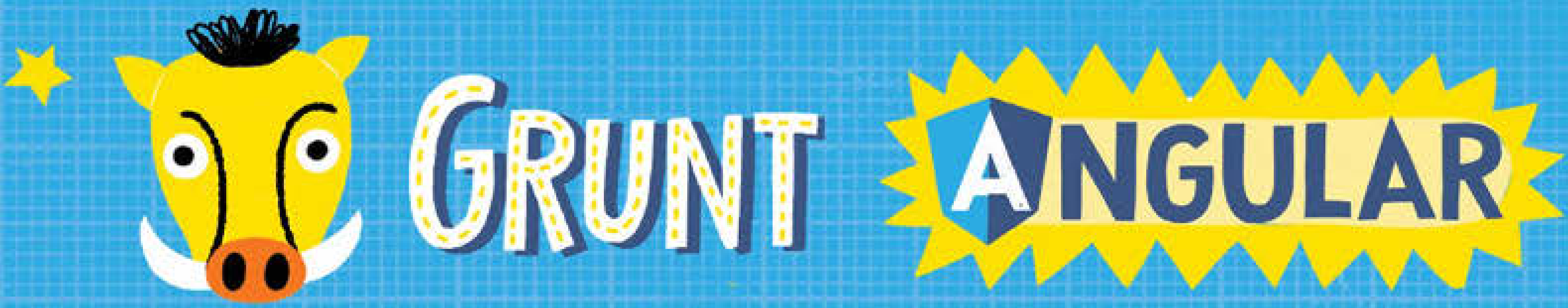
BROWSER SUPPORT

We feel it's important to inform our readers which browsers the technologies covered in our tutorials work with. Our browser support info is inspired by @andismith's excellent When Can I Use web widget (*andismith.github.io/caniuse-widget*). It explains from which version of each browser the features discussed are supported.



Author
JACK FRANKLIN
Jack is a developer at GoCardless, based in London. He writes regularly for the JavaScript Playground (*javascriptplayground.com*) and is often found playing with the latest library or tool

Illustration
LINZIE HUNTER
Linzie is a Scottish illustrator and hand-lettering artist based in London. Clients include *Time* magazine, the *Guardian*, Hallmark, Nike and the BBC
linziehunter.co.uk

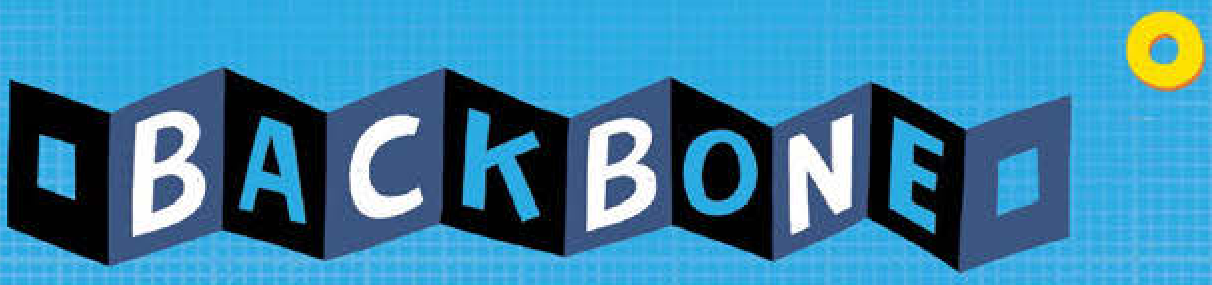


ESSENTIAL Javascript

The top 23 JavaScript libraries every designer and developer needs to know

The ecosystem of JavaScript libraries and tools is growing all the time, and getting to grips with what's available and what will work for your project can be difficult. In this article I've tried to list some of the best tools out there for different use cases and categories to give you a head start.

Remember, there's no real substitute for sitting down and trying out a tool, and I highly recommend doing that before committing to using a library on a project. Over time you'll come to establish your favourites and that's what you should aim for – a selection of libraries you trust and rely on. New toys to play with are great, but there's nothing better than going back to the ones you are comfortable with. ▶



When to use jQuery

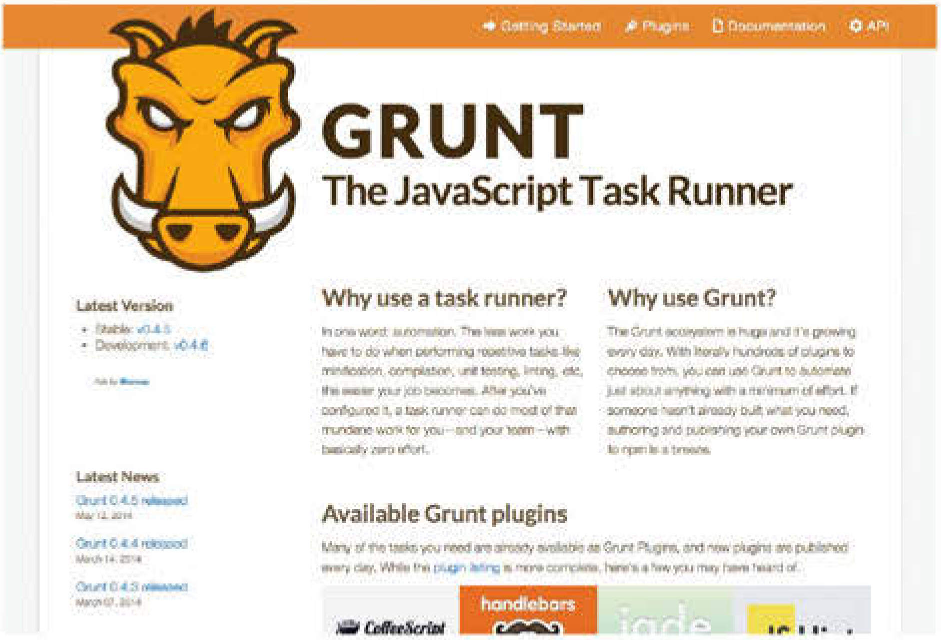
In 2013 Remy Sharp spoke at jQuery UK and wrote an accompanying blog post titled 'I know jQuery. Now what?' (*netm.ag/nowwhat-261*). In the post he spoke about how he went from relying on jQuery for all his projects, to using it on more of a case-by-case basis.

Tom Maslen, a developer at the BBC, spoke at Responsive Day Out (*besquare.me/session/cutting-the-mustard*) about 'Cutting the mustard', the technique the BBC use to decide if a browser is 'up to scratch' or not:

```
if ('querySelector' in document &&
  'localStorage' in window &&
  'addEventListener' in window) {
  // bootstrap the JavaScript application
}
```

Any browsers that don't support the above get the core experience, but without the extra functionality added by the JavaScript. If you need to fully support those older browsers, jQuery is still the best option.

If you find yourself only supporting the newer browsers but still wanting an easier interface than JavaScript offers by default, Remy Sharp's min.js (*github.com/remy/min.js*) might be the thing for you. It's a small library that offers simple DOM querying and event listener functionality.



Getting Started · Plugins · Documentation · API

GRUNT
The JavaScript Task Runner

Latest Version
- Stable: v0.4.5
- Development: v0.4.6

[illegible]

Latest News
Grunt 0.4.5 released
May 12, 2014
Grunt 0.4.4 released
March 14, 2014
Grunt 0.4.3 released
March 07, 2014

Why use a task runner?
In one word: automation. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting, etc, the easier your job becomes. After you've configured it, a task runner can do most of that mundane work for you—and your team—with basically zero effort.

Why use Grunt?
The Grunt ecosystem is huge and it's growing every day. With literally hundreds of plugins to choose from, you can use Grunt to automate just about anything with a minimum of effort. If someone hasn't already built what you need, authoring and publishing your own Grunt plugin to npm is a breeze.

Available Grunt plugins
Many of the tasks you need are already available as Grunt Plugins, and new plugins are published every day. While the plugin listing is more complete, here's a few you may have heard of.

GruntJS One of the frst and most popular projects for running command line tasks

COMMAND LINE TOOLS

Node.js

nodejs.org

Node.js lets us write JavaScript on the server, but it also lets us run arbitrary JavaScript on the command line. This has led to a number of tools designed to automate tasks that we otherwise would have to complete manually. The easy-to-use Node Package Manager (npm) has really helped the community and ecosystem grow at a rapid rate. It not only makes it very easy to install tools, but also makes it nice and straightforward for people to publish their own.

Grunt

gruntjs.com

Grunt has been around for a while now, and is the most popular task runner for JavaScript. Created by Ben Alman, its vast plugin ecosystem allows you to configure it to compile your Sass, run your tests, minify your JavaScript and much more. When using Grunt, you install the plugins for the functionality you desire, and then create a `Gruntfile.js` file which will contain all the configuration for the plugins – such as the location of files to use, or settings for how you would like your code to be minified. You don't write code to perform your tasks, but configure the plugins you'd like to use.

gulp.js

gulpjs.com

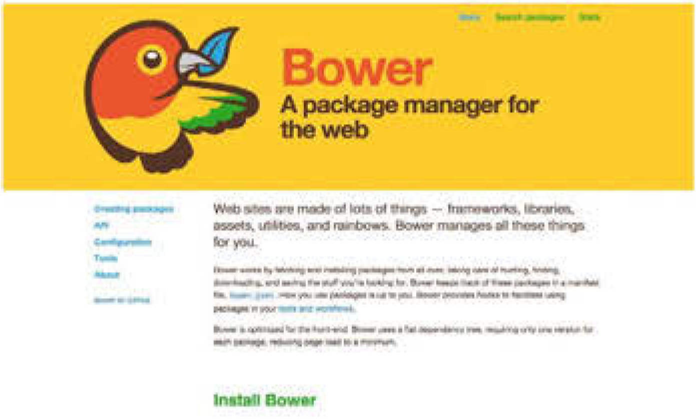
gulp.js uses file streams and lets you define tasks by writing the code, rather than the configuration approach favoured by Grunt. This means that you can get a bit more hands-on with your build system. In your `Gulpfile.js` file, tasks are defined as functions, in which you can write any JavaScript you'd like. There is still a huge amount of plugins for common tasks, but you have to include and call them yourself. It's a little bit more difficult to pick up, but once you're used to it you're able to wield a huge amount of power.

Serve

github.com/visionmedia/serve

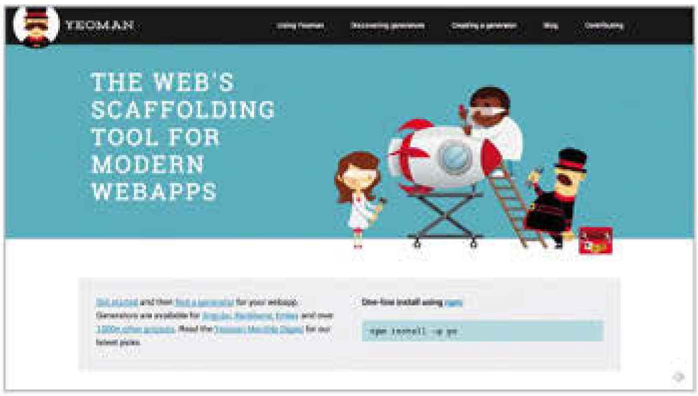
Serve is something I use every day and has become a key part of my workflow. Once installed, it allows me to run `serve` in any directory and visit the site on `localhost:3000`. This is fantastic if you're working with a website that you would quickly like to run, and view in a browser.

gulp This system focuses on streams, which are abstract representations of files



Web sites are made of lots of things — frameworks, libraries, assets, utilities, and rainbows. Bower manages all these things for you.

Bower For managing frontend dependencies, try Bower



```
npm install -g yo
```

Yeoman You can save time by using Yeoman to generate files for you

Parse, validate, manipulate, and display dates in javascript.

Moment.js This provides a fantastic API for editing and formatting dates

Yeoman

yeoman.io

Yeoman is a tool for building websites. It will generate a project structure for you, and configure Grunt or gulp to run your project. Once you're familiar with the tools, you can save a lot of time by using Yeoman to generate files for you. For example, running `yo webapp` will generate an entire project set-up for writing HTML, CSS (or if you prefer, Sass) and JavaScript, complete with a fully featured Gruntfile with tasks configured for you. You can also create custom generators to suit your project's particular needs.

Pulldown

github.com/jackfranklin/pulldown

I may be slightly biased here, as Pulldown is a tool that I built with Tom Ashworth, but it's one I use a lot for quickly downloading a library. It uses CDNJS (*cdnjs.com*) to search for files and then download them. This means that downloading jQuery becomes as quick as `pulldown jquery`. If you want more power, and more fine-grained dependency management, consider using Bower (see below) – but for a quick prototype or demo, I find Pulldown fits my needs really nicely.

Bower

bower.io

Bower is a fully featured solution for managing frontend libraries and dependencies in your projects. It can manage versions for you, and make sure that your project uses a specific version of a library. It will also deal with the dependencies of your dependencies, to ensure that every one works as hoped. You can keep your dependencies in a `bower.json` file – then running `bower install` will ensure all dependencies from `bower.json` are installed.

UTILITY LIBRARIES

Lo-Dash

lodash.com

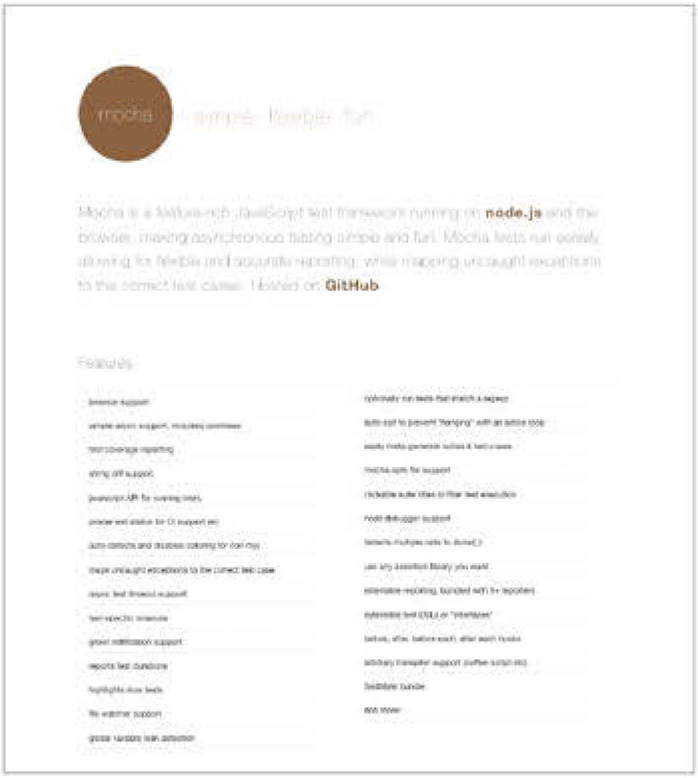
Although newer features in ECMAScript 5 and 6 have a lot of added functionality, often you'll find yourself wanting to perform a fairly simple operation on some data that you'd rather not implement yourself. That's where Lo-Dash comes in. It provides a vast amount of utility functions to perform common operations on objects, arrays or functions.

Moment.js

momentjs.com

Working with dates is usually pretty painful, regardless of language, but in JavaScript it's even worse. The provided APIs simply don't provide the features required. Thankfully for us, Moment.js exists to fill the gap. It's a library built to handle creating, manipulating

▶



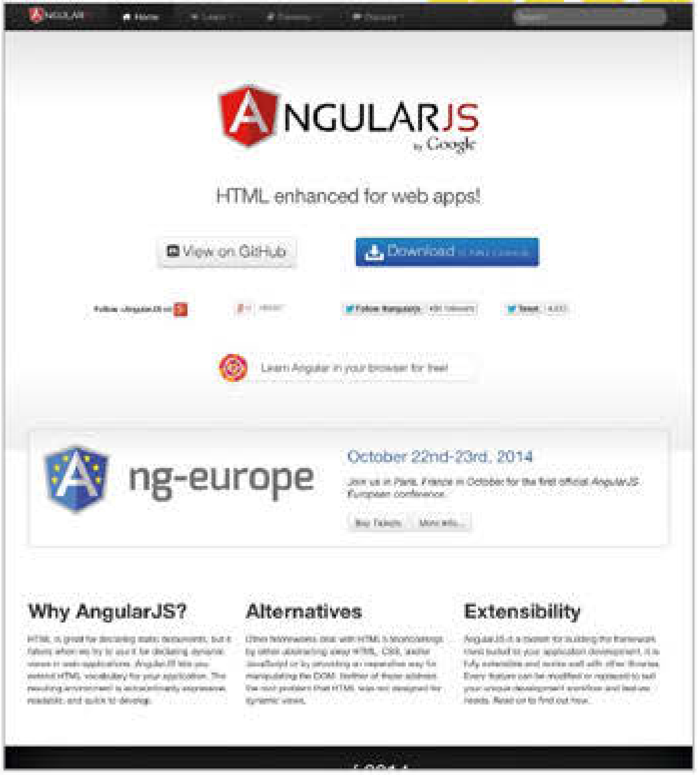
mocha simple, flexible, fun

Mocha is a feature-rich JavaScript test framework running on **node.js** and the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases. Hosted on **GitHub**.

Features

- browser support
- simple async support, including promises
- test coverage reporting
- string diff support
- javascript API for running tests
- proper exit status for CI support etc
- auto-detects and disables coloring for non-ttys
- maps uncaught exceptions to the correct test case
- async test timeout support
- test-specific timeouts
- growl notification support
- reports test durations
- highlights slow tests
- file watcher support
- global variable leak detection
- optionally run tests that match a regexp
- auto-exit to prevent "hanging" with an active loop
- easily meta-generate suites & test-cases
- mocha.opts file support
- clickable suite titles to filter test execution
- node debugger support
- detects multiple calls to done()
- use any assertion library you want
- extensible reporting, bundled with 9+ reporters
- extensible test DSLs or "interfaces"
- before, after, before each, after each hooks
- arbitrary transpiler support (coffee-script etc)
- TextMate bundle
- and more!

Mocha What sets this testing framework apart is that it doesn't provide an assertion library



HTML enhanced for web apps!

October 22nd-23rd, 2014

Join us in Paris, France in October for the first official AngularJS European conference.

Why AngularJS?

HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications. AngularJS lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.

Alternatives

Other frameworks deal with HTML's shortcomings by either abstracting away HTML, CSS, and/or JavaScript or by providing an imperative way for manipulating the DOM. Neither of these address the root problem that HTML was not designed for dynamic views.

Extensibility

AngularJS is a toolset for building the framework most suited to your application development. It is fully extensible and works well with other libraries. Every feature can be modified or replaced to suit your unique development workflow and feature needs. Read on to find out how.

AngularJS This offers tools for creating your own web application framework

and formatting dates and incredibly feature-complete. It's become something I instantly reach for the moment I have to work with dates.

Accounting.js

netm.ag/AccountingJS-261

Have you ever tried to multiply two decimal numbers in JavaScript?

```
0.2 * 0.1
>0.020000000000000004
```

Because of how numbers are stored internally, these small errors are common, and in a large application they can start to cause huge problems later on, that are really difficult to track down. If you're working with any form of currency or just numbers in general, you owe it to yourself to use Accounting.js and save many hours of pain.

TESTING LIBRARIES

Testing has gone from something that was never really talked about five or so years ago (at least, not in the JavaScript world) to being something you can't avoid. When I learned JavaScript, the idea of writing tests for my code never occurred to me, but now I can't imagine writing code without tests. It gives you confidence that what you've written works, helps with the design of code, and prevents you from inadvertently breaking something in the future.

Jasmine

netm.ag/Jasmine-261

Jasmine is one of the most popular testing frameworks for JavaScript. It lets you structure your tests through the use of `describe` functions, and specify the expected behaviour with `it` functions. It also provides easy ways to set expectations, such as `expect(2+2).toEqual(4);`. It runs in the browser and comes with plenty of advanced functionality, including spies for those with more complex applications to test.

QUnit

qunitjs.com

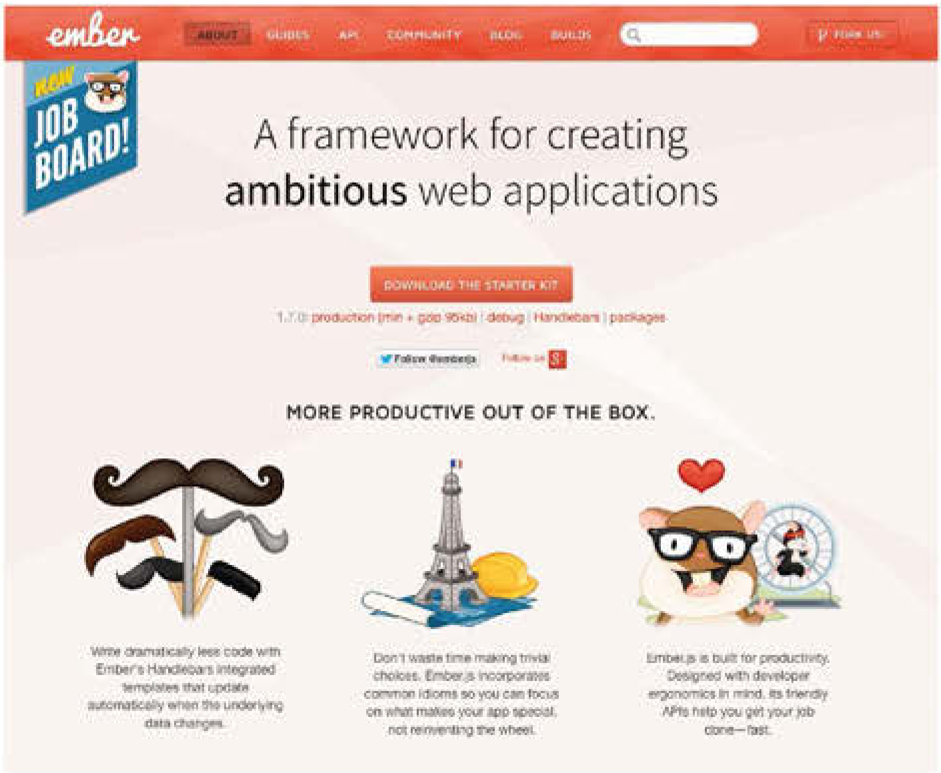
QUnit is the library that jQuery is tested in, so you can be confident it provides all the functionality you need. QUnit is particularly useful when you want to test code that modifies DOM elements. It lets you define HTML that will be set up for each test, and then reset before the next one. This means you can run code and assert that the HTML has been manipulated as expected. If you're working on JS that interacts heavily with the DOM, QUnit is the best choice.

Mocha

visionmedia.github.io/mocha

Mocha is a testing framework that can run in the browser, or alternatively within a Node.js application. Similarly to Jasmine, it provides functions including `describe` and `it`, and a number of other features that you would expect from any good testing framework.

What differentiates Mocha, however, is that it does not provide an expectation or assertion library. This means that within your `it` blocks, you're able to use any library to make your actual test assertions. The ability to use the expectation framework of choice, along with the fact that Mocha supports both



A framework for creating ambitious web applications

1.7.0: production (min + gzip 95kb) | debug | Handlebars | packages

MORE PRODUCTIVE OUT OF THE BOX.

Write dramatically less code with Ember's Handlebars integrated templates that update automatically when the underlying data changes.

Don't waste time making trivial choices. Ember.js incorporates common idioms so you can focus on what makes your app special, not reinventing the wheel.

Ember.js is built for productivity. Designed with developer ergonomics in mind, its friendly APIs help you get your job done—fast.

EmberJS An opinionated framework, EmberJS is full of conventions to offer a framework for building complex web apps

the browser and Node, makes it a very popular choice.

SINGLE-PAGE APPLICATION FRAMEWORKS

Due to the advances in browser capabilities and the functionality of JavaScript (along with future enhancements that aren't quite there yet), in recent times more and more people have taken to writing their applications on the client. This means typically the server just sends data to the client in the form of JSON through an API. The rendering of that data, along with all other logic, is written in JavaScript in the client. As the need for these applications has grown, so have the frameworks to support them.

Ember
emberjs.com

On its homepage, Ember describes itself as "a framework for creating ambitious web applications", and it focuses heavily on conventions to help you build your applications quickly. It puts the URL first, ensuring that if a user refreshes a page in your JS app, they get that same page back again in exactly the same state, which is a common problem with JavaScript applications.

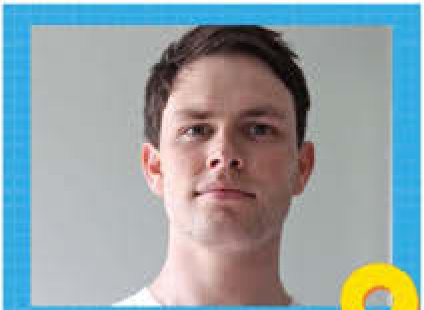
AngularJS
angularjs.org

AngularJS is Google's framework for building web applications, and takes a very different approach to Ember. It provides much more freedom in that it lets you do things exactly how you want – which can be both a blessing and a curse.

The best way to use Angular is to define your own internal conventions on how you want to structure your app, or you'll find as your app grows it will become very difficult to work with. Todd Motto's style guide (*netm.ag/motto-261*) may be a good place to start.

BackboneJS
backbonejs.org

BackboneJS is much less fully featured than the heavyweights of Ember and AngularJS, but that's entirely by design. ▶



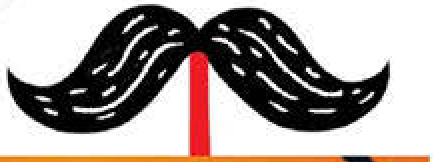
Q&A Andy Appleton

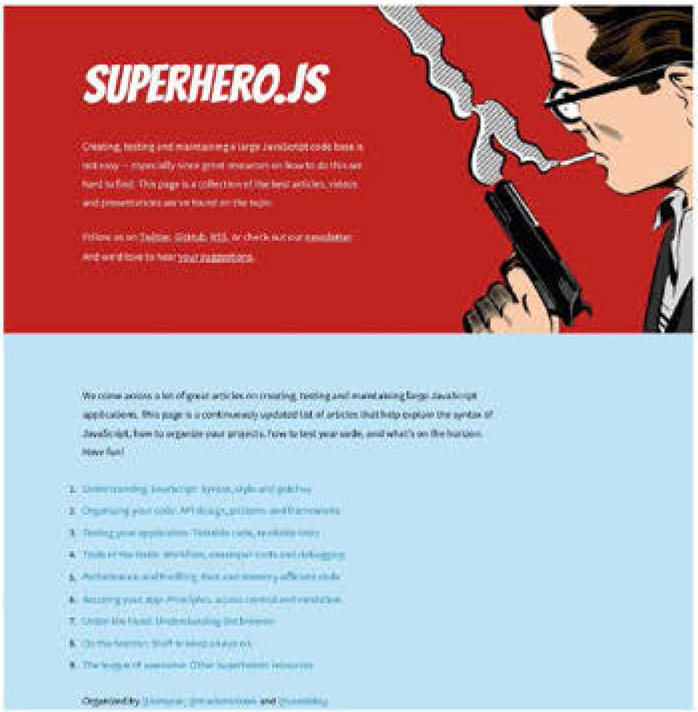
Andy Appleton (@appltn), developer at Heroku, on the JS tools he relies on and the future releases he's excited about

Are there any tools that you tend to rely on for every project?
Node.js and npm, always. Regardless of whether the backend is written in JavaScript, I always end up using Node in some capacity. I almost always include Lo-Dash (*lodash.com*). I like the expressiveness of the functional code I can write with it, and it's nice to use as a compatibility layer when working with array methods like `forEach` and `map`.

Which of the single-page application libraries do you tend to pick?
The library I'm most familiar with is AngularJS. Angular is great for building applications with a complex UI, and once you wrap your head around directives, they're an extremely powerful way to isolate and abstract interaction. I've also been working with Ember. I think Ember with ember-data (*github.com/emberjs/data*) is a really good fit for a data-heavy app, and ember-cli (*ember-cli.com*) brings a strong set of conventions which make it far easier to work on a project with a larger team.

Are there any libraries in their infancy you're excited about?
Projects like Hoodie (*hood.ie*) and Meteor (*meteor.com*) are extremely interesting. I'm of the opinion that the next massive progression in the way we build web applications will be in handling offline syncable data. I think about how Ruby on Rails has made it easy to create RESTful APIs, and I want that for data sync too.





SUPERHERO.JS

Creating, testing and maintaining a large JavaScript code base is not easy — especially since great resources on how to do this are hard to find. This page is a collection of the best articles, videos and presentations we've found on the topic.

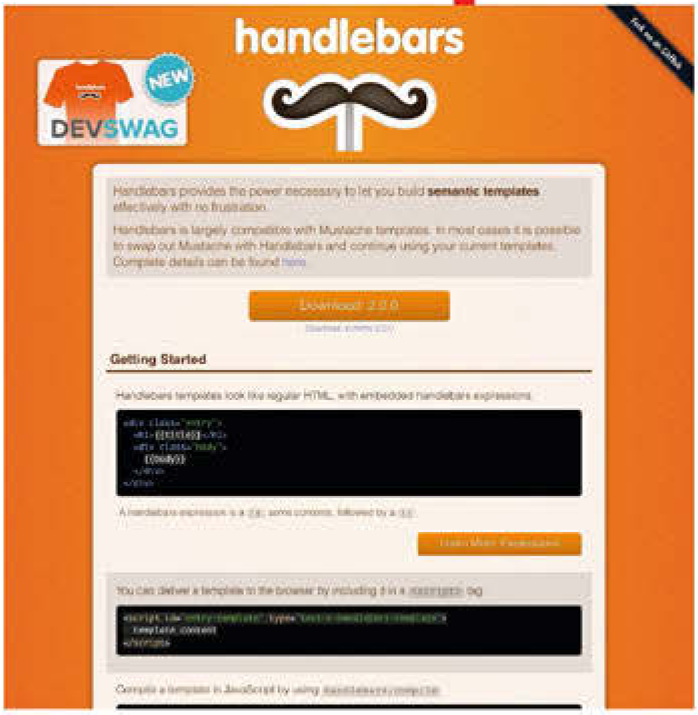
Follow us on Twitter, Github, RSS, or check out our newsletter and we'd love to hear your suggestions.

We come across a lot of great articles on creating, testing and maintaining large JavaScript applications. This page is a continuously updated list of articles that help explain the syntax of JavaScript, how to organize your projects, how to test your code, and what's on the horizon. Have fun!

1. Understanding JavaScript: Syntax, style and pitfalls
2. Organizing your code: API design, patterns and frameworks
3. Testing your application: Testable code, available tools
4. Tools of the trade: Workflow, essential tools and debugging
5. Performance and Profiling: Fast and memory efficient code
6. Securing your app: Principles, access control and validation
7. Under the Hood: Understanding the browser
8. On the horizon: Stuff to keep an eye on
9. The league of awesome: Other superhero resources

Organized by [illegible], [illegible] and [illegible]

SuperheroJS A curated collection of articles, books, talks and tutorials all about JavaScript



Handlebars provides the power necessary to let you build **semantic templates** effectively with no frustration.

Handlebars is largely compatible with Mustache templates. In most cases it is possible to swap out Mustache with Handlebars and continue using your current templates. Complete details can be found here.

Getting Started

Handlebars templates look like regular HTML, with embedded handlebars expressions.

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

A handlebars expression is a {{, some contents, followed by a }}

You can deliver a template to the browser by including it in a <script> tag.

```
<script id="entry-template" type="text/x-handlebars-template">
  template content
</script>
```

Compile a template in JavaScript by using Handlebars.compile

Handlebars The templating language that comes baked in with Ember

Further reading

There's so much out there that no article could hope to cover even half of what's available, so below I've shared some further links and resources to help.

SuperheroJS *superherojs.com*
A great suite of curated articles, tutorials, videos, books and conference talks to help you improve your JavaScript and learn what the future holds for the language, too.

EchoJS *echojs.com*
EchoJS is a site where people submit JavaScript content which is then upvoted by the community. It's a great place to find tutorials and new libraries that people release.

JavaScript Weekly
javascriptweekly.com
A weekly, curated list of content in your inbox, JavaScript Weekly and its corresponding Twitter account (@JavaScriptDaily) offers a great way to get a dose of JavaScript.

What Backbone gives you is a lower-level set of components that you can build your application on top of. It gives you models, collections and views, but lets you choose anything else to fill the gaps. This does mean you have to write a lot more code yourself, but it gives you complete control.

ReactJS
facebook.github.io/react

It's probably a little unfair to have React in this list, as it's not a framework for building these large applications. Rather, it provides a view layer for applications, which can be used with any other framework. React focuses on re-rendering states into views and doing it in the most efficient way, with the fewest possible DOM manipulations. It does this by building a virtual DOM, comparing it to the real one in the page, and making them match.

TEMPLATING

With the growth in the use of JavaScript for rendering content on the frontend, templating languages have sprung up to support this.

Handlebars
handlebarsjs.com

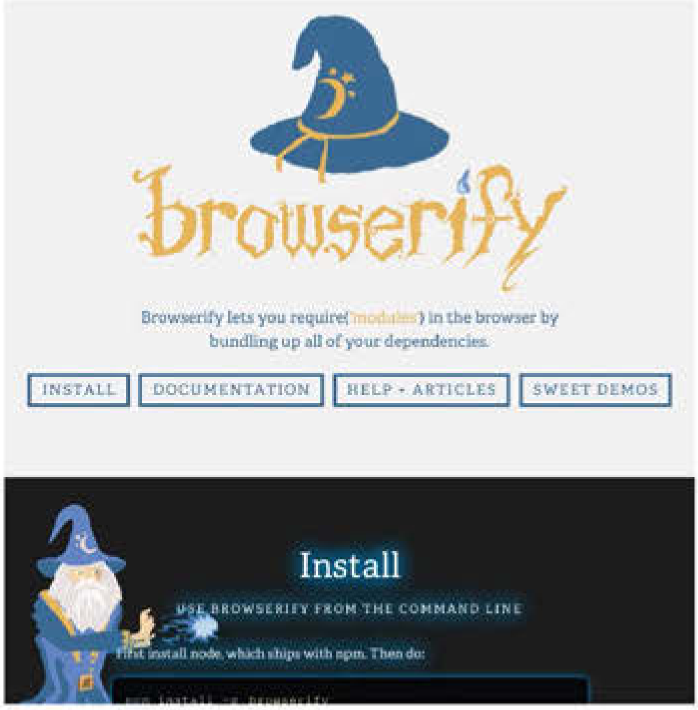
If you use EmberJS you'll use Handlebars: it's the templating language that comes baked in with Ember, but it can also be used independently. Handlebars uses curly brackets for syntax, and supports basic logical structures such as conditionals and loops to help you output your data.

Lo-Dash templates
lodash.com/docs#template

Lo-Dash, the utility library mentioned earlier, comes with a small and simple templating engine built in. If you find yourself needing some small templates, – perhaps without some of the more advanced features that other solutions offer – Lo-Dash may well be the library for you.

MODULES AND DEPENDENCY MANAGEMENT

As JavaScript apps continue to get larger and the number of files grows, you need to make sure you use some form of system for managing the files, the



Browserify lets you require('modules') in the browser by bundling up all of your dependencies.

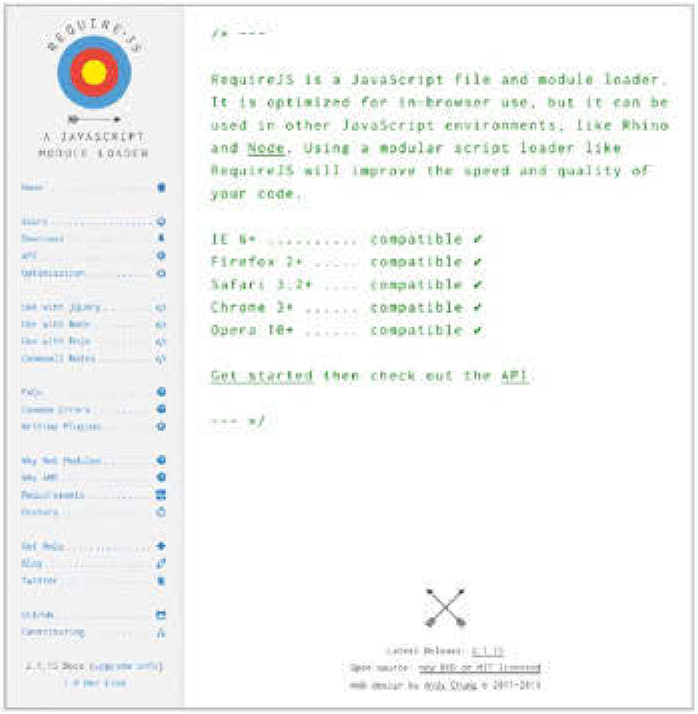
Install

USE BROWSERIFY FROM THE COMMAND LINE

First install node, which ships with npm. Then do:

```
npm install -g browserify
```

Browserify This lets you use NodeJS's modules and npm to manage frontend dependencies



```
/* ---

RequireJS is a JavaScript file and module loader.
It is optimized for in-browser use, but it can be
used in other JavaScript environments, like Rhino
and Node. Using a modular script loader like
RequireJS will improve the speed and quality of
your code.

IE 6+ .......... compatible ✔
Firefox 2+ ..... compatible ✔
Safari 3.2+ .... compatible ✔
Chrome 3+ ...... compatible ✔
Opera 10+ ...... compatible ✔

Get started then check out the API.

--- */
```

RequireJS For working with Asynchronous Module Definition spec, this is the best choice

dependencies between them, and how they are loaded into a page. Having thousands of `<script>` tags in a page is not a valid approach here!

Browserify

browserify.org

Browserify takes the module system we're lucky to have in Node with npm, and lets us use it on the frontend. You write all your code using CommonJS (the dependency system in Node) and use npm to manage dependencies. Before viewing it in a browser, you then run it through Browserify, which converts it into code that the browser can happily understand. The downside is that you will need to rerun Browserify every single time you change a file, but this is easily automated and can be done with Grunt, gulp or similar.

jspm.io

jspm.io

Created by Guy Bedford, jspm is a package manager for JavaScript that lets you get modules from a number of sources, including npm, GitHub and others.

RequireJS

requirejs.org

RequireJS is the best framework for working with the Asynchronous Module Definition spec for loading and managing dependencies. You define and load in your modules in a very specific format, which lets you also define your own modules and their dependencies.

All the dependencies are loaded asynchronously, which means that, unlike tools like Browserify, you have to define your module in a callback function. It's a different way of working and it can take some time to get your head around things, but once you have it's a really nice way to build applications.

ES6 Modules

netm.ag/polyfill-261

Coming soon to a browser near you, ECMAScript 6 – the latest version of JavaScript – will provide a full implementation for creating, importing and exporting code as modules. It's not supported to a degree where you could reasonably use it, but the ES6 Module Loader polyfill provides support for using it today. If you'd like to get up to speed with the module system that will eventually be supported in browsers, this is the way to go. The polyfill is very thorough and usable; I've used it in a large Angular application to manage dependencies with very little grief.

CONCLUSION

The JavaScript world is vast, and full of great libraries and tools to make your workflow and development easier and more efficient. However, there's so much out there that it can often be difficult to filter down to the tools you need for your daily work.

The approach I recommend is to try a variety of the tools I've listed that sound like they might suit you, and slowly whittle the list down to just the final few that you can learn in detail and come to rely on for most of your projects. There are new libraries released all the time, but don't be tempted to chop and change your toolset all the time. Find the ones you like and stick with them. Occasionally, maybe try a new one and see if it can do a better job than your existing tool, but there's nothing wrong with having your favourites. ◘

THE Complete WEB DESIGN TOOLKIT

WE REVEAL THE PERFECT WORKFLOW TO ELIMINATE FRONTEND DEVELOPMENT PAIN

Part 1.

In the first of a two-part series, **Adam Simpson** and **Neil Renicker** tackle frontend development pain and staying ahead of the curve

It would be impossible to document every frontend tool and workflow in an article … and even if we did, it would be overwhelming and a bit discouraging. We decided to outline a comprehensive tool belt, kitted out with some of the best equipment, and share a workflow that gives the tools meaning and context. These are definitely not the only right answers, but we've found them invaluable.

LET'S TALK ABOUT EDITORS

As a frontend developer, a huge percentage of your time is spent in your editor. Since we take pride in the code we author, configuring a proper editor is the first step.

THE BIG TWO

There are dozens of text editors to choose from, but we want to focus in on two: Sublime Text (*sublimetext.com*) and Vim (*vim.org*). Before you protest that we left off your favourite editor, hear us out.

Our hope is that you take the time to learn the dark corners of your editor. Only then will you begin to alleviate pain points and become more productive.

LET'S TAKE A LOOK AT SUBLIME

Sublime Text, our first recommendation, is available on all three major operating systems, and it's blazing fast. Better yet, it has a thriving plugin ecosystem, so it's infinitely customisable. Here are a few essential Sublime plugins:

AUTHORS

ADAM SIMPSON
(@a_simpson) is a frontend developer at Sparkbox (*seesparkbox.com*); he's passionate about semantic markup, crafting quality code and creating accessible experiences. His personal site is at *adamsimpson.net*

NEIL RENICKER
Neil (*@neilrenicker*) is a converted print designer who brings that sensibility to his role as a frontend dev at Sparkbox. He writes about working on the web at *neilrenicker.com*

ILLUSTRATION

TOBIAS HALL
Tobias (@tobiashall) is an illustrator, designer, mural artist and hand-letterer extraordinaire working out of London (*tobias-hall.co.uk*)

STATIC DESIGN TOOLS

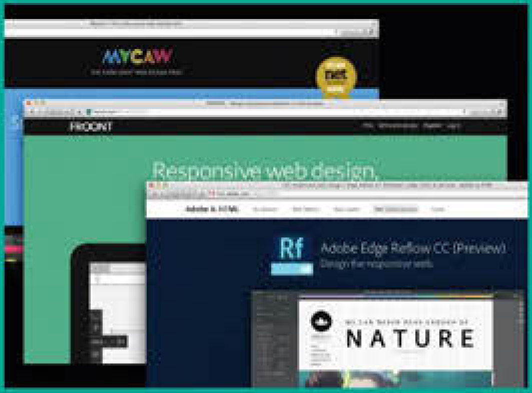
As frontend developers, we're often involved in the design of the projects we build. Many designers still love Photoshop and Illustrator for their design work, but there are are some great newcomers worth considering (see ***netm.ag/4tools-255*** for more on this). Our favourite new tool for web and UI design is Sketch (***bohemiancoding.com/sketch***). It's a vector-based design tool akin to Illustrator, but lighter, and intended for the web. Common UI design tasks such as rounding corners and adding drop shadows are instantly accessible, and exporting your work is effortless.

Professional digital design for Mac.

Sketch gives you the power, flexibility and speed you always wanted in a lightweight and easy-to-use package. Finally you can focus on what you do best: Design.

Fine lines The Sketch homepage (bohemiancoding.com/sketch)

Keep your eye on these too: Froont (*froont.com*), Macaw (*macaw.co*) and Adobe Edge Reflow (*html.adobe.com/edge/reflow*). These apps all let designers work in an environment closer to the code. A tool that generates code is a great place to start for creating design mockups, and might give you a head start on projects. But be careful: don't let your tools generate all your code under the covers. You're responsible for what you create, so you'll feel better if you know what it's doing!



New kids Fresh blood in the design tools environment

- ● Package Manager (*sublime.wbond.net*)
- ● Emmet (*emmet.io*)
- ● SidebarEnhancements (*github.com/titoBouzout/SideBarEnhancements*)
- ● GitGutter (*github.com/jisaacks/GitGutter*)

VIM, SUBLIME'S UGLY COUSIN

Vim is our second choice because it has a much steeper learning curve than Sublime: you might think of it as being the next level up. The power of Vim lies in its system of movements and macros. If you want to dive in and start understanding Vim's craziness, head over to your local command line and run `vimtutor`.

After you've progressed through vimtutor, you can start cultivating your own .vimrc file. The .vimrc is the configuration file for Vim, and is the place where you define all of your plugins and customisations. A couple well worth checking out are *github.com/amix/vimrc* and *github.com/mutewinter/dot_vim*, which just so happens to be from our co-worker, Jeremy Mack (@mutewinter).

PICK THE RIGHT TOOL

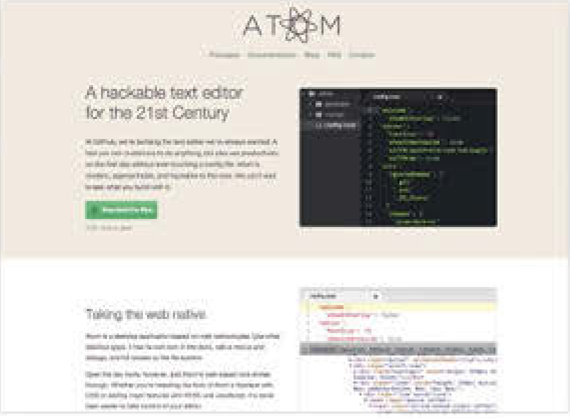
If neither Sublime nor Vim makes sense for you, here's a list of perfectly acceptable alternatives:

- ● Coda (*panic.com/coda*)
- ● Espresso (*macrabbit.com/espresso*)
- ● Atom (*atom.io*)
- ● TextMate 2 (*github.com/textmate/textmate*)

As always, be ready to defend your decision, but be willing to admit it if you run into a superior tool.

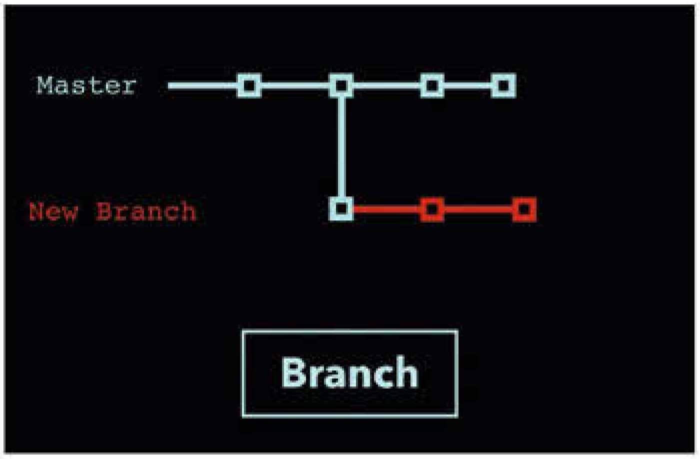
PROTECT YOUR CODE WITH VERSION CONTROL

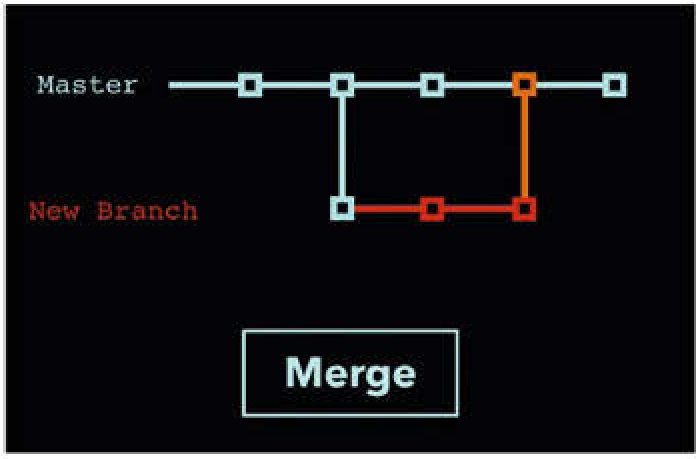
Writing code is hard enough. Collaboration with other developers is harder yet. Typical collaboration tools fail when applied to code: email, USB drives, mounted network drives,



A hackable text editor for the 21st Century

Taking the web native.





Below left Github's Atom editor (*atom.io*) shows promise as an extensible, comfortable editor

Above left Illustration of a branch in a version control system (VCS)

Above right Illustration of a merge in a VCS

and Dropbox shares are all great for sharing pictures, movies, or presentations. However, these tools offer no solution to the problem of multiple developers modifying the same codebase at the same time. There has to be a better way.

Version control to the rescue

Version control is "a system that records changes to a file or set of files over time so that you can recall specific versions later" (*git-scm.com/book/en/Getting-Started-About-Version-Control*).

> WRITING CODE IS HARD ENOUGH. COLLABORATION WITH OTHER DEVELOPERS IS HARDER YET

Git (*git-scm.com*), Mercurial (*mercurial.selenic.com*) and Subversion (*subversion.apache.org*) are all version control systems (VCS). They enable a group of devs to seamlessly contribute to the same codebase.

Let's learn some terminology

'Pull' is a download from a remote version control server. 'Push' is an upload to a remote server. 'Branch' is a complete duplicate of the main code line. 'Merge' is taking a branch and merging its changes into another branch. A 'conflict' is when the software cannot automatically merge two changesets together and you have to manually complete the merge. Now that we're all speaking the same language, let's dive in.

The perfect source control workflow

We use Git and a remote Git server such as GitHub (*github.com/features*) or Bitbucket (*bitbucket.org*). We love Git's effortless branching, blazing speed and superb reliability. We use GitHub or Bitbucket because they offer incredible features such as issue tracking, a place for discussions about the code, wiki hosting, and a Git remote server.

Our workflow involves creating a branch for every change to the codebase. These are commonly called 'feature branches'. If you want to contribute code to a project, clone the Git repository and create a new branch off the master branch. This new branch is your sandbox: your changes do not have any effect on the master branch. This enables you to experiment with the code – you can break things as much as you like. This feature branch is your own little version of the project.

Once you are ready to share your hard work with the rest of your team, create a 'pull request' on GitHub. A pull request is a GitHub feature for merging code into master. We typically take this moment as an opportunity to peer review the work before merging in the new code. To recap, our workflow boils down to these steps:

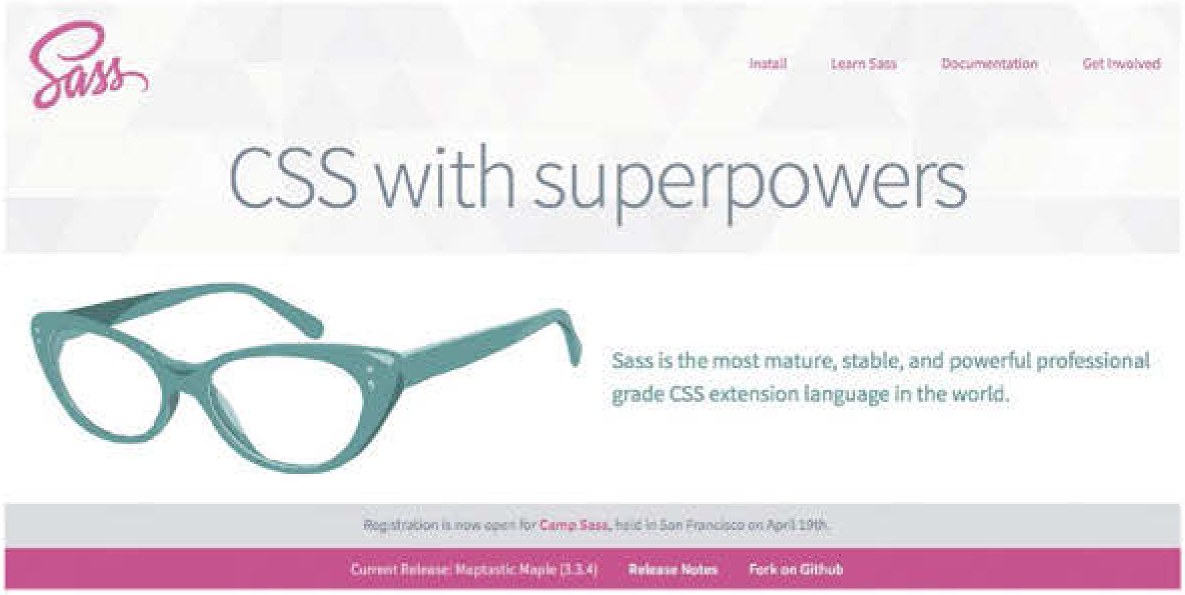
1 Create a new branch from master
2 Make changes to that branch
3 Push to GitHub/Bitbucket and create a pull request
4 Review the pull request with a co-worker
5 Merge the branch into master

This workflow will make you more productive and safeguard your code. Go branch crazy; enjoy the flow.

Preprocessing: end manual labour

Computer languages have always evolved. This constant evolution is why we love our industry.

►



CSS with superpowers

Sass is the most mature, stable, and powerful professional grade CSS extension language in the world.

Registration is now open for Camp Sass, held in San Francisco on April 19th.

Current Release: Maptastic Maple (3.3.4)

Change is exciting, but HTML, CSS and JavaScript have remained static for years. Consider this:

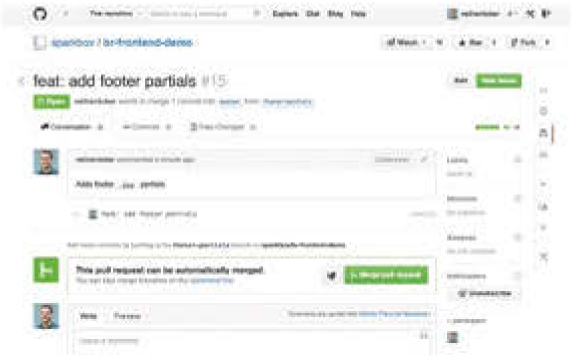
- HTML is an often redundant markup language
- CSS has a 'flat' styling syntax
- JavaScript is weird

That's where preprocessing comes in – if you've been holding off, there's never been a better time to jump in. Preprocessors are high-level languages that improve upon those we already have, adding better syntax and missing features. The code then gets compiled into its native language. Preprocessors let us live in a fantasy world where we can augment these native languages as we see fit.

The important ones to focus on as a frontend developer are CSS and JavaScript preprocessors. Here's what's available to you:

CSS Preprocessors

- Less (*lesscss.org*)
- Stylus (*learnboost.github.io/stylus*)
- Sass (*sass-lang.com*)



feat: add footer partials #15

This pull request can be automatically merged.

Above Among CSS preprocessors, Sass has gained broad appeal

Right A pull request viewed on GitHub

Top right Preprocessors improve on the languages we already have, adding better syntax and features

The choice of which CSS preprocessor you use is entirely up to you. It's worth spending some time in the documentation of each of the projects and determining which one fits your requirements. You can also try all these flavours through such services as CodePen (*codepen.io*) or jsBin (*jsbin.com*).

We prefer Sass (the .scss variant), mostly because it has gained broad appeal, and it still feels a lot like CSS. Sass has always felt like CSS 2.0 to us: it's comfortable and familiar, but it has superpowers that CSS can't even fathom.

JavaScript preprocessors

In the realm of JavaScript preprocessors, CoffeeScript (*coffeescript.org*) is the most popular. In CoffeeScript, functions are clean and beautiful, most punctuation is optional, and conditional statements can be written in plain English.

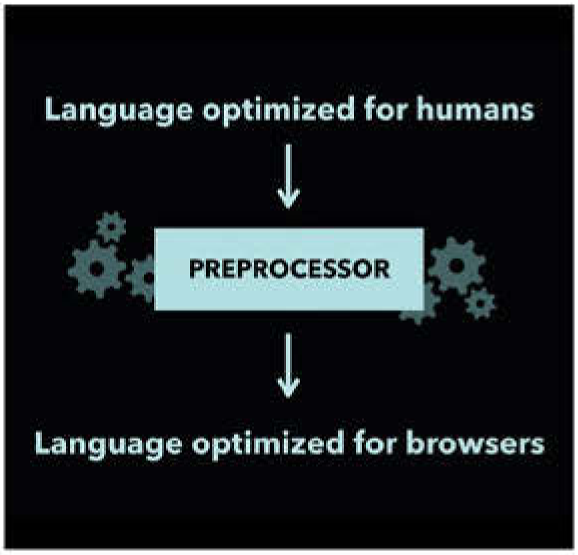
How do I get started with preprocessing?

You may want to consider applications such as CodeKit for Mac (*incident57.com/codekit/index.html*) or Prepros for Windows and Mac (*alphapixels.com/prepros*). These are third party applications aimed at making preprocessing dead simple. If you're working on your own, or on a really small team, they might be all you need.

It's challenging, however, to work on a distributed team using a third party application. Dealing with versioning and environments becomes a real pain point. That's where open source software comes in.

Open source task runners

Open source task runners can run local servers, minify/concat/lint our code, condense images, and even refresh our browser when we save new code.



Language optimized for humans

PREPROCESSOR

Language optimized for browsers

Here's the best part about open source task runners: they include text-based configuration files that can be checked into source control. This ensures that every developer working on the code can work in the same environment and has access to the same third party libraries.

Getting started with task runners

There are several open source task runners that could do the job. Grunt (*gruntjs.com*) is the most popular. Gulp (*gulpjs.com*) is a promising newbie, and some folks (*gist.github.com/toolmantim/6200029* and *algorithms.rdio.com/post/make*) are using established software such as Make for asset compilation and task running.

We recommend starting with Grunt: the getting started guide (*gruntjs.com/getting-started*) should have you up and running. For a thorough, beginner-friendly approach, try Chris Coyier's piece 'Grunt for People Who Think Things Like Grunt are Weird and Hard' (*24ways.org/2013/grunt-is-not-weird-and-hard*).

Using a tool such as Grunt will be a big boost to your workflow. There's a lot of power in it, and once

IF YOU'VE BEEN HOLDING OFF PREPROCESSORS, THERE'S NEVER BEEN A BETTER TIME TO JUMP IN

▶

THE OPEN SOURCE STACK

You're probably noticing that many of the tools we recommend are open source projects. If open source is new to you, you'll need to take a bit of time to get your machine set up to run this software. Here's what you need to get going:

GIT

- Mac: Install Git (***git-scm.com/downloads***) or download GitHub app (***mac.github.com***)
- Windows: Install Git for Windows (***msysgit.github.io***) or download the GitHub app (***windows.github.com***)

RUBY

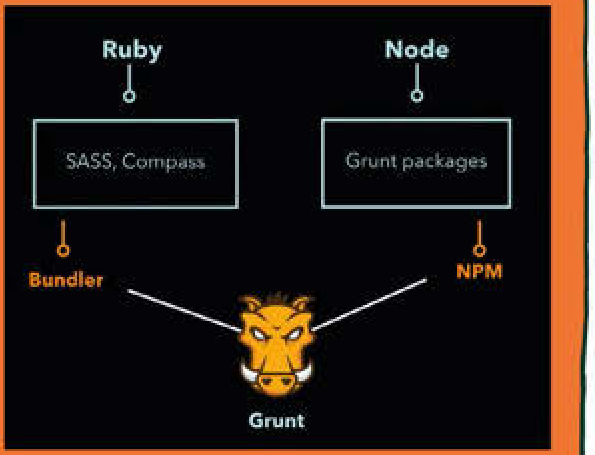
- Mac: Use the system version of Ruby in 10.9 or install Ruby using RVM (***rvm.io***)
- Windows: Install Ruby (***rubyinstaller.org***)
- Install Bundler (***bundler.io***)

NODE.JS & GRUNT

- Install Node.js (***nodejs.org***)
- Install Grunt.js (***gruntjs.com/getting-started***)

For more detailed installation guides, here are a few links to get you started:

- Mac OS X Dev Setup (***github.com/nicolashery/mac-dev-setup***)
- Love Your Frontend Tools on Windows (***seesparkbox.com/foundry/love_your_frontend_tools_windows***)



All stacked up Illustration of the open source stack

→ Getting Started Plugins Documentation API

GRUNT

The JavaScript Task Runner

Latest Version

- Stable: v0.4.4

Discover Dev Tools, a free screencast to help you master Chrome Dev Tools.

Why use a task runner?

In one word: automation. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting, etc, the easier your job becomes. After you've configured it, a task runner can do most of that mundane work for you – and your team – with basically zero effort.

Why use Grunt?

The Grunt ecosystem is huge and it's growing every day. With literally hundreds of plugins to choose from, you can use Grunt to automate just about anything with a minimum of effort. If someone hasn't already built what you need, authoring and publishing your own Grunt plugin to npm is a breeze.

Right Grunt (*gruntjs.com*) is the most popular open source task runner

▶ you have mastered the basics, you'll find you never want to go back.

Modular markup and styles

What's wrong with HTML?

Learning to write HTML isn't a daunting task: you only need to know a few HTML tags to get started. But as time goes on, your once-simple markup can become difficult to manage. Approaching markup from a modular perspective is a huge step forward. Fortunately, we have tooling to make this possible.

Get modular with static site generator

Writing modular markup isn't a new concept. Content management systems such as WordPress, Drupal and ExpressionEngine have this idea built in: write boilerplate code once, and then repeat it in the future as often as you need.

Static site generators (*staticsitegenerators.net*) are intended to help developers automate the management of sites without the overhead of a database. The frontend development community has co-opted them as a tool for creating static templates (HTML, CSS, and JavaScript) even if the final destination is a web app or complex CMS.

How do you choose your static site generator?

If you want minimal set-up time, and you work solo or on a small team, you might be interested in a proprietary static site generator. Hammer (*hammerformac.com*) is a Mac application that will enable you to get started quickly in a friendly native environment. Cactus (*cactusformac.com*) is a brand new Mac app promising similar benefits. These applications are essentially good-looking wrappers around a templating language.

A few open source projects worth consideration are Jekyll (*jekyllrb.com*) and Middleman (*middlemanapp.com*). We're smitten with a newcomer to the party, Assemble (*assemble.io*), which is flexible enough to facilitate making decisions about the way we want to structure our project, but opinionated enough to keep us from spinning our wheels.

Understanding Git Stash

Git may feel overwhelming at first, but once you're comfortable with its basic structure you can begin to appreciate its power. One of our favourite 'power user' features is Git Stash. To understand Git Stash, you have to understand the concept of clean or dirty states in your repository. A clean state has no uncommitted changes; you have saved everything, and Git knows about all your changes. A dirty state is the opposite: there are changes you haven't committed yet. Moving from a dirty state to a clean state is the soothing rhythm of a Git repo.

In Git Stash you have an alternative that encourages experimentation and problem solving. Let's say you made some changes. You want to commit the good ones, but you don't want to lose the half-baked ones. Stash swoops in to the rescue. Run `git stash` in your project: now your repo is in a clean state. Your changes can be re-applied from the stash at any time. You can even view all your stashes by typing `git stash list`.

We hope you consider adding Stash into your workflow; it's magical! To dive deeper, head over to the documentation (***git-scm.com/book/en/Git-Tools-Stashing***).

Super-charge your web development with Hammer

Still using PHP includes for HTML? You're going to love Hammer

BUILD RIGHT AND

SET YOUR SITE FREE

For the fluidity of your content, for the integrity of your vision, for the cause of a more durable web.

Rooted in the American heartland, our team of builders answers the call for a better web. Driven by deep conviction in a web liberated from device constraints, Sparkbox leads the way toward a greater standard as we write, educate, and build.

Far left Hammer (*hammerformac.com*) is a Mac app that acts as a first step towards templating and modular markup

Left The new Sparkbox website (*seesparkbox.com*) is built entirely as a static site with Assemble, with the exception of the blog! More info at *seesparkbox.com/foundry/static_ee*

Getting Started With Assemble

We've created a demo project (*github.com/sparkbox/br-frontend-demo*) exploring how Assemble works.

Assemble enables you to separate your site into three major components: layouts, data, and partials. Layouts are like themes, or page types. Data is your content: stuff that can change regularly. Partials are markup snippets, reused across your pages.

> MODULAR MARKUP ISN'T A NEW CONCEPT: WRITE BOILERPLATE ONCE, REPEAT AS OFTEN AS YOU NEED

Most static site generators have these notions, with different ways of handling and naming them. In the above demo, you can see this separation. We consign layouts, partials and data to their own special homes.

Living the dream: modularise all the things

This finely tuned structure facilitates working efficiently. Instead of repeating the same code, we can iterate over each piece of data in our data file:

```
{{#each data-item }}
  {{> _partial }}
{{/each}}
```

There! We have programming power with our markup. Better yet, no content is written in the markup. It's all generated from a separate data file.

Let's make our CSS modular too! We can now give our CSS files the same name as our HTML module names. Take a look at the the `/scss` directory in the demo project, and compare it with the markup files in `templates/partials`. We've modularised our markup into bite-sized chunks, and our styles match them. You can imagine the long-term organisation and maintainability benefits this structure gives you.

What now?

Using Assemble isn't the Holy Grail. We've found it useful, but these same principles also apply to your preferred static site generator. Remember, we're looking for these three elements of separation:

1 Layouts
2 Partials
3 Data

If you adopt this modular mindset, your projects will become more organised and scalable.

Let's wrap this up

That's a lot to digest: once you get serious about tooling, it can seem you've taken a step towards complexity. There's no need to bite all of this off at once. In Part 2 we'll explore local servers, productivity, device testing, browser testing, and automated deployment. Stay tuned!

These tools aren't magical; exploring them is crucial. If one doesn't fit your team, discard it and keep looking. If you see room for a tool that doesn't exist, stand on the shoulders of the open source giants and make one to share with the community! ■

Thanks to Jody Thornburg (@jodytburg) for her advice and assistance in reviewing this article

THE Complete WEB DESIGN TOOLKIT

AUTHORS

ADAM SIMPSON
Adam (@a_simpson) is a frontend developer at Sparkbox (*seesparkbox.com*). His personal site is *adamsimpson.net*

NEIL RENICKER
Neil (@neilrenicker) is an ex-print designer turned frontend dev at Sparkbox. Read more at *neilrenicker.com*

ILLUSTRATION

TOBIAS HALL
Tobias (@tobiashall) is an illustrator, designer, mural artist and hand-letterer extraordinaire (*tobias-hall.co.uk*)

PART 2

In the second of our two-part toolkit series, **Adam Simpson** and **Neil Renicker** reveal the tools they use to streamline those vital but time-consuming tasks

Last time, we covered the frontend tooling stack from top to bottom – but there's still plenty more to learn. Sometimes the biggest pain points for frontend developers are on the fringes of our workflow. It's the little things that crop up throughout the life of a project that can be the most difficult. This article will explore these pain points, and consider how modern computing and frontend tooling can be used to alleviate them.

PRODUCTIVITY

Have you ever thought: "All I'd need is one extra hour per day, and I could get so much more done"? It's tempting to think that adding extra hours would magically make us more productive, and free us up to do the things we love. Well, you're in luck – incorporate a few productivity hacks into your workflow as a frontend developer, and you might get that extra hour after all.

At its core, productivity is really about being lazy ... the good kind of lazy. Do you know the acronym DRY? It stands for Don't Repeat Yourself, and it's an adage you should take to heart. As you analyse your processes, you'll probably find a lot of areas where you could optimise your workflow and spend less time spinning your wheels.

Take control of the command line

We've seen a lot of tweets recently about whether or not a frontend developer should learn to use the command line. Our answer is, "Absolutely yes". Sure, you can be a talented web worker and never crack open Terminal, but we believe you're missing out on a lot of productivity gains.

Start by learning basic navigation commands: `cd`, `ls` and `pwd`. From there, you can start to learn to manipulate files, install developer packages and run build systems. If you're just getting started, give *linuxcommand.org* a try. For an even gentler start, we like John W. Long's article, The designer's guide to the OSX command prompt (*netm.ag/prompt-256*). When you're ready to go deeper, you might enjoy our article on the Sparkbox Foundry: 'Command Line Tools for Frontend Developers' (*netm.ag/commandline-256*).

Stop clicking so much

Using your mouse can be cumbersome. Sure, sometimes it's just the best way to navigate an interface, but for everyday tasks, it can really slow you down. Here are a few of our favourite apps for transitioning away from the mouse and towards the keyboard:

Alfred (*alfredapp.com*) is a free Mac application that gives you incredible power in your keyboard for actions like launching apps, finding files, and even running custom scripts or workflows. The single best piece of advice we can offer for boosting your productivity is to go and download Alfred right now. Or if you're on a Windows machine, you might give Launchy (*launchy.net*) a try.

Dash (*kapeli.com/dash*) is another of those apps we can hardly imagine working without. It gives you offline access to code documentation. While that might not sound like much, it's worth every penny.

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Alfred Hide / Show Hidden File Extentions

Posted: November 14th 2012, 10:30:47 am
Read Time: 0 minutes, 51 seconds

Alfred is one of the most useful tools I think I own on my Mac it speeds up production speeds like nothing you've ever seen. One of my biggest gripes with Mac OS

ALFRED WORKFLOWS

Alfred (*alfredapp.com*) gives you incredible keyboard control over actions like launching apps, finding files, and even running custom scripts or workflows. Here are a few of our favourite Alfred workflows:

- **GitHub** (*netm.ag/alfred-github-256*) Search and launch GitHub repos with a few keystrokes.
- **Pinboard** (*netm.ag/alfred-pinboard-256*) Search for your bookmarks saved on Pinboard (*pinboard.in*).
- **Browser tabs** (*netm.ag/alfred-tabs-256*) Search through open tabs in Chrome or Safari and jump to a particular one.
- **Colours** (*netm.ag/alfred-colours-256*) Type 'c' into Alfred, or paste in a hex colour value, then launch that colour in the system colour picker to manipulate it. You can also get alternate syntaxes of the same colour: paste in a hex value, then copy out the RGB value, for example.
- **Harvest** (*netm.ag/alfred-harvest-256*) Start and stop your timers and add notes to the Harvest time-tracking app.
- **Can I use...** (*caniuse.com*) Search from Alfred to *caniuse.com* to determine browser support for specific web APIs and CSS features.
- **Show hidden file extensions** (*netm.ag/alfred-hidden-256*) Show or hide all the invisible files in Finder: overwhelming, but sometimes necessary.

▶ Dash is completely based on keyboard commands, and it's blazingly fast. We've never had so much fun browsing code documentation.

But here's the best part: Dash has a wonderful workflow for Alfred that enables you to instantly search code documentation from anywhere on your system. Look for the Alfred workflow in Dash's `Preferences>Integration` pane.

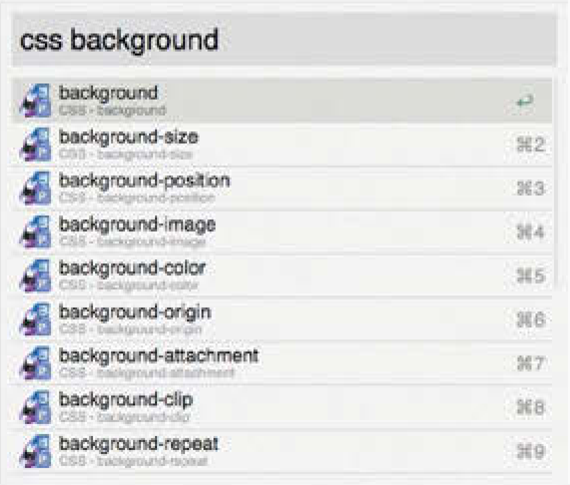
Tools for handling text If you're itching for even more productivity apps, here are a few of our favourite tools for dealing with text:

- Markdown (*daringfireball.net/projects/markdown*) enables you to quickly write semantic documents that can be compiled to HTML.
- nvAlt (*brettterpstra.com/projects/nvalt*) is a scratchpad and repository for your writing. It includes Markdown support, great keyboard shortcuts, and the ability to quickly search all your notes.
- TextExpander (*smilesoftware.com/textexpander*) creates a library of short keywords that expand into longer words or phrases. We use it for tasks like typing our email and mailing addresses, code snippets, and any phrases we use a lot.

LOCAL SERVERS

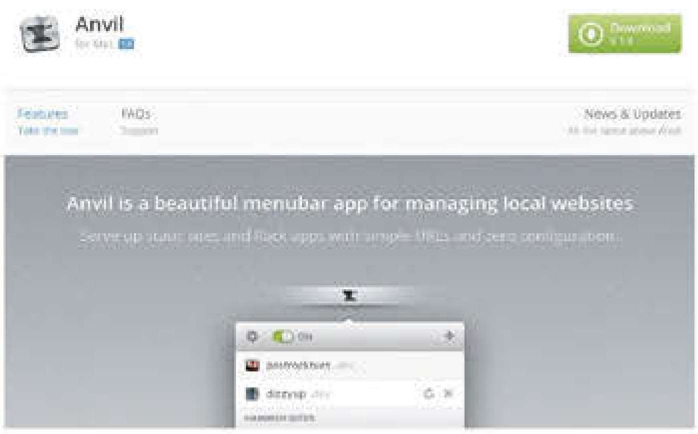
Pushing changes to a remote server many times per day creates lots of pain. Working locally drastically shortens the time between making a change in your code and seeing the change reflected in your browser, thus enabling you to iterate quickly and experiment much more freely.

There are several software packages that help make setting up a local development environment much simpler. One of our personal favourites is Anvil (*anvilformac.com*), a Mac app that sits in your

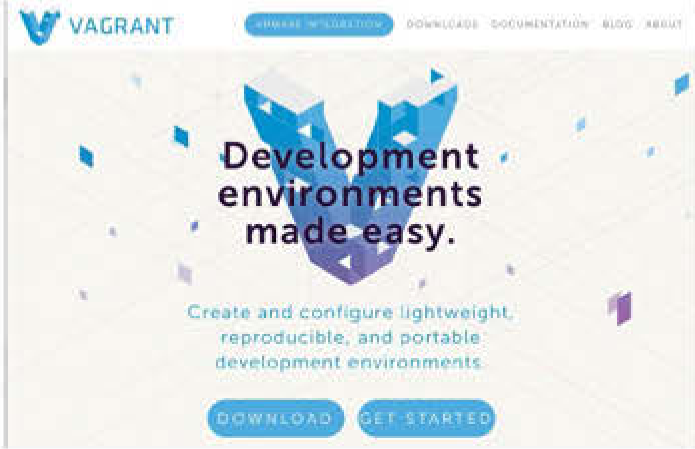


css background

background CSS - background	↩
background-size CSS - background-size	⌘2
background-position CSS - background-position	⌘3
background-image CSS - background-image	⌘4
background-color CSS - background-color	⌘5
background-origin CSS - background-origin	⌘6
background-attachment CSS - background-attachment	⌘7
background-clip CSS - background-clip	⌘8
background-repeat CSS - background-repeat	⌘9


Anvil is a beautiful menubar app for managing local websites

Serve up static sites and Rack apps with simple URLs and zero configuration.


Development environments made easy.

Create and configure lightweight, reproducible, and portable development environments.

Far left Dash stores snippets of code and searches offline documentation sets for over 130 APIs. It integrates with keyboard shortcut tool Alfred

Above left Vagrant lets you create a standard local development environment for you and everyone on your team

Above right Anvil makes setting up a local development environment a breeze, giving you a local server and .dev domains

menu bar. It gives you a local server and pretty .dev domains.

There are also more fully featured tools that provide server and database software to run CMSs and apps. MAMP (*mamp.info*) and WampServer (*wampserver.com*) are two such packages. Available for Mac and Windows respectively, these packages provide a complete Apache, MySQL, and PHP environment. If you like having more control over your local server configuration, you can roll your own environment by installing Apache, MySQL and PHP from source, which enables you to mirror a production environment on your local machine.

YOUR BROWSER IS A DESIGN TOOL. THAT'S NOT SOMETHING YOU HEAR EVERY DAY …

A fantastic plugin that we're using in our Grunt process is grunt-contrib-connect (*netm.ag/grunt-256*). This plugin spins up a basic HTTP server and enables live reloading of your pages. Sweet!

One new approach is to use a Virtual Machine (VM) to clone your production server configuration to your local machine. Services such as Docker (*docker.io*) and Vagrant (*vagrantup.com*) provide ways to accomplish this. Both packages give you text-based configuration files that reliably and predictably spin up VMS-matching criteria you specify, and enable you to check that configuration into your version-control system. These solutions require more set-up time, but may provide the ultimate local development experience.

Working locally may require an organisational change or a temporary step backwards in efficiency while you get all the pieces set up locally, but ultimately, it'll be worth it.

BROWSER AND DEVICE TESTING

Remember those days when you had to test your JavaScript by writing alerts?

```
alert("JavaScript is loaded!");
```

Hopefully, this isn't your first stop for debugging JavaScript these days, but it reminds us of a time when tooling for testing browser code was archaic and unfriendly.

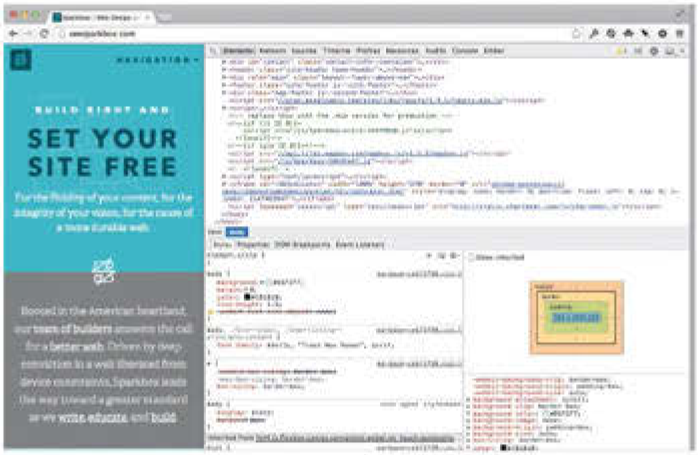
In 2006, Firebug (*getfirebug.com*) was released. Browser debugging tools existed before then, but Firebug tied the tools together and made significant improvements to them. The next year, Microsoft shipped a developer toolbar compatible with IE6 and IE7. Finally, in 2009, Microsoft shipped IE8 with developer tools built in.

As you can see, we haven't had excellent browser debugging tools for that long, really. It's easy to forget how good we have it now.

The browser as design tool

Your browser is a design tool. That's not something you hear every day, is it? We're starting to think that way, though. Our modern browsers are equipped with such sophisticated developer tools that they are no longer the last stop in our development process. Instead, they are starting to become one of the first.

Many of us began our foray into the web via traditional graphic design. Naturally, we approached the web the way we approached advertising: starting by designing a series of static, high-fidelity mockups that our clients could review. ▶


NAVIGATION
BUILD RIGHT AND
SET YOUR
SITE FREE
Elements Network Sources Timeline Profiles Resources Audits Console

This page, left Setting your browser developer tools' Inspector as a vertical pane makes it easier to check responsive designs

Right Need help with Chrome DevTools? Code School has an excellent free course (*discover-devtools.codeschool.com*)

Facing page, left Beanstalk lets you deploy from a source-control repository to a remote host, and works well for simple deployments

Right For complex app deployments, Mina lets you build and run scripts on servers via SSH

▶ Over time, though, we've begun to realise that working in such a fashion just doesn't complement the fluid, rapidly changing nature of the web. And with advanced tooling giving us fine-grained control over our work, right there in our browsers, we have few excuses left not to start working in the new way.

Using developer tools

First things first: view the source of your web page. We love Chrome's DevTools, so we will focus on Chrome here. Most of these principles apply across browsers, although the terminology may differ slightly. On the Mac, hit `Cmd+Alt+I` to open the browser's developer tools and view the source. You can also right-click on any element in your web page, then select `Inspect Element` to open the Inspector and focus on that element.

Once you open up your inspector, you'll be greeted with the friendly Elements tab. This shows the structure of your HTML page, and enables you to explore and edit your DOM. Did you know you can drag DOM elements around, edit them by right-clicking, and delete them by pressing your `Delete` key? This is starting to feel like a design tool!

Just underneath the Elements tab, you'll find the Styles pane. The CSS that applies to your currently selected DOM node will appear in this pane. Here's the best part, though: you can edit these styles in real time – not unlike the way you would in a design tool – delete old styles or add new ones. In Chrome's developer tools, it's good to know that you can select the CSS value, then simply use your Up and Down arrow keys with variations of Shift and Alt to make incremental changes to numeric values.

We love using a layout where the Inspector sits to the right of the browser. This enables you to expand and contract the Inspector pane, which flexes your website for testing responsive designs. Most browsers offer a menu item to toggle this view.

Digging deeper

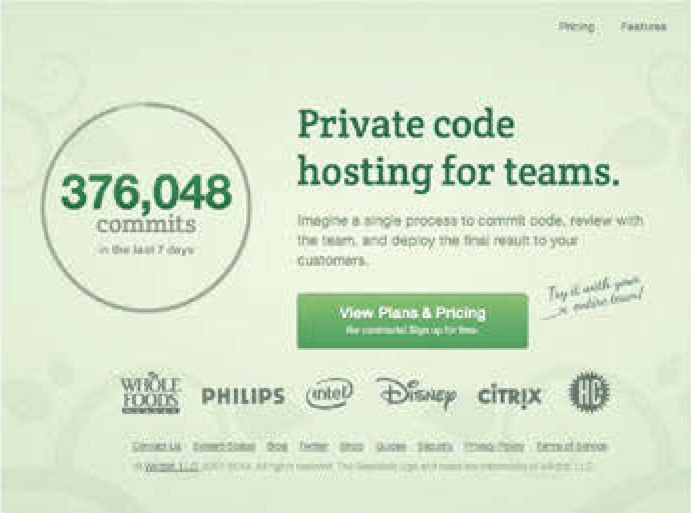
There's a lot more to explore in Chrome's developer tools. Spend time tinkering with the Resources tab, the Console tab for debugging JavaScript, and the Audits panel for getting a handle on your site's performance. We recommend reading the Chrome DevTools documentation (*netm.ag/devtools-256*) and watching CodeSchool's excellent free course (*discover-devtools.codeschool.com*).

AS YOU ENCOUNTER PAIN POINTS, TAKE A MOMENT AND TRY TO IMPLEMENT A MORE GRACEFUL SOLUTION

Automated deployment

Manually dragging and dropping, copying and pasting, or selecting and moving files is not a great way to deploy your project. What if you forget one hidden file that holds your app variables? Or if someone else has to work on the deployment? Will they know what to do?

The tools we have discussed so far make developer tasks repeatable, predictable, and store them in source control. We can and should approach deployments with the same ideals.

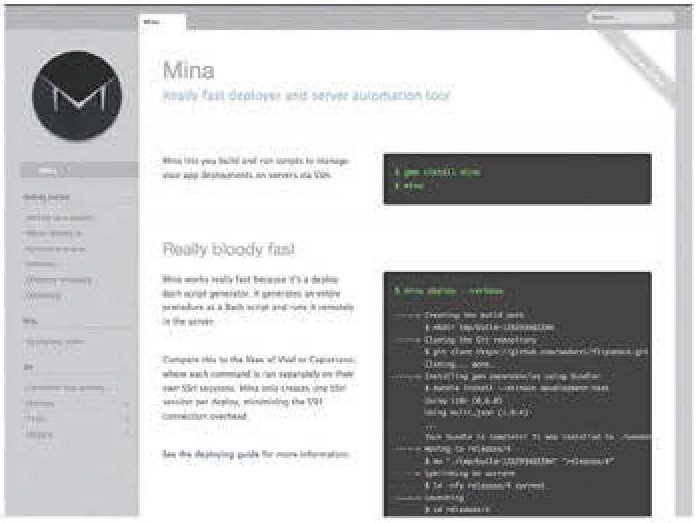


Pricing Features

376,048 commits in the last 7 days

Private code hosting for teams.

Imagine a single process to commit code, review with the team, and deploy the final result to your customers.



Mina

Really fast deployer and server automation tool

Mina lets you build and run scripts to manage your app deployments on servers via SSH.

```
$ gem install mina
$ mina
```

Really bloody fast

Mina works really fast because it's a deploy Bash script generator. It generates an entire procedure as a Bash script and runs it remotely in the server.

Compare this to the likes of Vlad or Capistrano, where each command is run separately on their own SSH sessions. Mina only creates one SSH session per deploy, minimizing the SSH connection overhead.

See the deploying guide for more information.

```
$ mina deploy --verbose
-----> Creating the build path
       $ mkdir tmp/build-[illegible]
-----> Cloning the Git repository
       $ git clone https://github.com/nadarei/flipstack.git
       Cloning... done.
-----> Installing gem dependencies using Bundler
       $ bundle install --without development:test
       Using i18n (0.6.0)
       Using multi_json (1.0.4)
       ...
       Your bundle is complete! It was installed to ./vendor
-----> Moving to releases/4
       $ mv "./tmp/build-[illegible]" "releases/4"
-----> Symlinking to current
       $ ln -nfs releases/4 current
-----> Launching
       $ cd releases/4
```

There are several services, like Beanstalk (*beanstalkapp.com*), FTPloy (*ftploy.com*) and Deploy (*deployhq.com*) that enable you to deploy from a source-control repository to a remote host. These work great for simple deployments. However, we've found that we need more control and more functionality in our deployments.

We've used the open-source projects Capistrano (*capistranorb.com*) and Mina (*nadarei.co/mina*) to automate our deployments. These projects are similar in their functionality, with a few trivial differences. At their core, their tasks are written in Ruby, and send the server a Bash script generated from the Ruby tasks. You can think of them as pre-processors for Bash.

Deployment programs like Capistrano and Mina will Deployment programs like Capistrano and Mina will use SSH (Secure Shell, a standard secure method for remote logins) to access your server and git-clone your project. Once the program clones the project, it then proceeds to perform whatever scripted actions have been written in the deployment configuration file. Automated deployments eliminate the pain we've all felt when manually dragging and dropping files onto an FTP server – let the computer do that hard work for you.

LET'S WRAP THIS UP

The biggest lesson to take away from our exploration of frontend tooling is that you shouldn't be afraid to explore. There's no need to adopt every tool at once. As you encounter pain points, take a moment and try to implement a more graceful solution. Don't know a more graceful solution? Do some research. And as you learn, be sure to share your findings with the rest of our community.

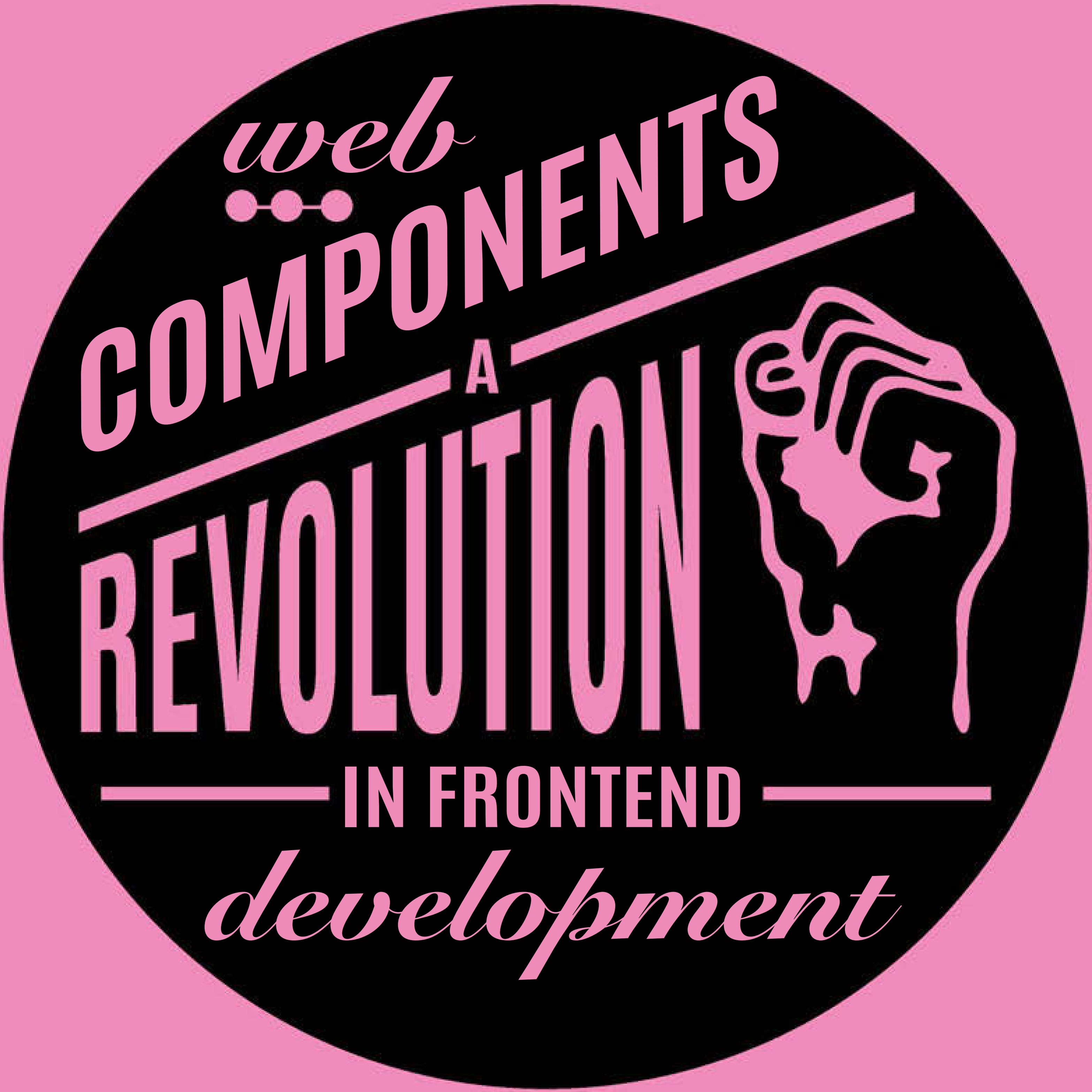
BROWSER DEVICE TESTING

Ah, device testing. It's one of the most important parts of our job as frontend developers. So how do we make it less painful? For most of us, relying on physical devices isn't practical. That's where virtualisation comes in. Virtualisation enables us to simulate other operating systems so we can do the testing on our local machine.

Here's a rundown of our favourites:

- Use VMWare (*vmware.com*) or VirtualBox (*virtualbox.org*) on your Mac to test in Internet Explorer. You can download free short-term Windows VM licences at *modern.ie*, or install them with one command in your terminal (learn how at *netm.ag/testing-256*)
- Use BrowserStack (*browserstack.com*) to do testing on virtual machines for almost every device you could imagine. It's all right there in the browser, so set-up is effortless
- Use Xcode's built-in iOS simulator for testing mobile versions of iOS, and Google's Android Emulator (*netm.ag/androidemulator-256*) for testing your designs on Android

We also really love Adobe Edge Inspect (*netm.ag/edge-256*) and Ghostlab (*vanamco.com/ghostlab*) for setting up synchronised testing on your mobile devices. Device testing is a very important problem to solve. Don't be afraid to invest some time and money in a sorting out a set-up you're really happy with.



web components: a revolution in frontend development

Web Components will revolutionise the way we develop sites, argues **Peter Gasston**. But what are they? Here's an outline of the basics

Code reusability is the Holy Grail of frontend web development: to create blocks of code that can be used across different sites without refactoring. The most common approach is to create a pattern of markup that can be copied and pasted into different documents, or included with a client or server-side templating language. Alternatively, there are many libraries of predefined patterns, like jQuery UI, that require that content is marked up in a certain way to take advantage of them.

There are a few problems with these approaches, chief among them being naming conflicts: how do you prevent CSS (or JavaScript) from the page being applied to your component, or vice versa? There are many methodologies aimed at decoupling markup from presentation and behaviour – notably in CSS with approaches like SMACSS and BEM – but while they are definitely steps in the right direction, they're still not foolproof.

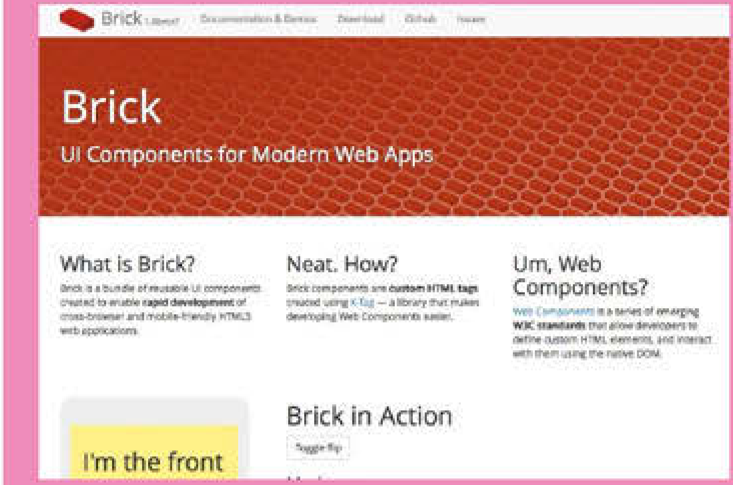
This notion of total two-way independence of a component and a page is known as 'encapsulation' and is common in object-oriented programming languages. It actually exists already in HTML in the form of `iframe`s, but `iframe`s have drawbacks: an extra network request for each, plus multiple rendering contexts, means a hit to page performance. There is, however, a better way – and browser vendors have been doing it for years, but not letting you know about it. Consider the `video` element. Its on-screen controls are built with HTML and CSS, but not exposed to you. The code for these controls is completely self-contained; the markup is hidden from the DOM, the CSS isn't in the CSSOM, and `querySelector()` won't be able to select it. If you view the source of a `video` element, all you will see is the `video` element.

THE FUTURE: WEB COMPONENTS

This is the core concept behind Web Components: fully encapsulated blocks of code that you can drop into any web

▶

AUTHOR
PETER GASSTON
Peter (*broken-links.com*) is creative technologist at Rehabstudio. He's a veteran developer and speaker, and is the author of *The Book of CSS3* and *The Modern Web*



Brick Library? Documentation & Demos Download Github Issues

Brick

UI Components for Modern Web Apps

What is Brick?

Brick is a bundle of reusable UI components created to enable **rapid development** of cross-browser and mobile-friendly HTML5 web applications.

Neat. How?

Brick components are **custom HTML tags** created using X-Tag — a library that makes developing Web Components easier.

Um, Web Components?

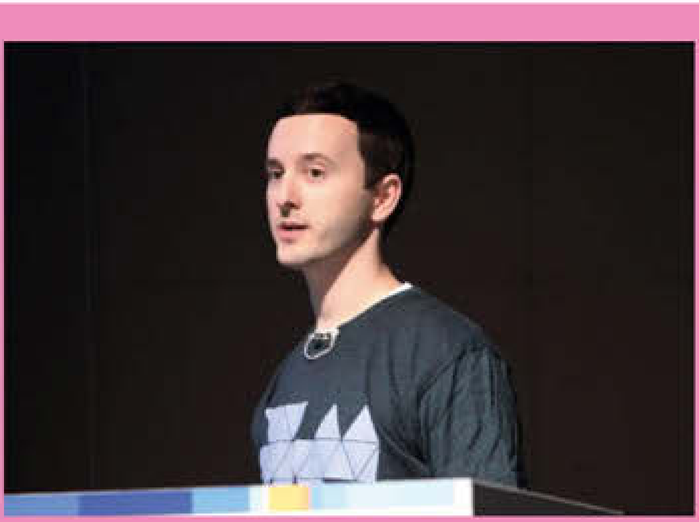
Web Components is a series of emerging **W3C standards** that allow developers to define custom HTML elements, and interact with them using the native DOM.

I'm the front

Brick in Action

Toggle flip

Building blocks The Brick UI library (*mozilla.github.io/brick*) is a useful cross-browser resource



Key player Google's Eric Bidelman (*github.com/ebidel*) is a key proponent of Web Components

page and have them look exactly the same, free of conflicts with, or inheritance from, existing code. At least, that's what the promise is.

More prosaically, 'Web Components' is a convenient collective name for a suite of new technologies that work towards a common purpose but also function individually. Some are new and some are improvements to existing properties. But they all work together to provide something genuinely new and useful.

In this article, I'm going to explore some of the technologies that make up Web Components, across HTML, CSS and JavaScript. As these are experimental properties and subject to change I'll try to keep code to a minimum, instead focusing mainly on concepts. And I'll start with the technology that's been around the longest – albeit hidden away, and not formally named until recently.

Put simply, the 'Shadow DOM' is the DOM hidden inside elements with on-screen controls. It's outside the regular DOM and inaccessible to JavaScript and CSS except by explicit invitation (I'll come to that shortly).

SHOW SHADOW DOM

I mentioned the code for these elements isn't visible by viewing the source of the document, but that's not actually correct; there is a way to see it. In Chrome, open DevTools, enable the `Show Shadow DOM` option in the settings. View the source again. Inside the video element you'll see a new text label, `#document-fragment`, and inside that, the DOM for the controls.

Creating your own Shadow DOM tree inside an element is easy. First, choose the element to which you want to append the Shadow DOM; this element is called the 'shadow host' and in this example I'm going to use a button:

```
<button>Go</button>
```

The next step is to create a new root, known as the 'shadow root' (basically, everything related to the Shadow DOM has the word 'shadow' prepended to it); this is done with the `createShadowRoot()` method:

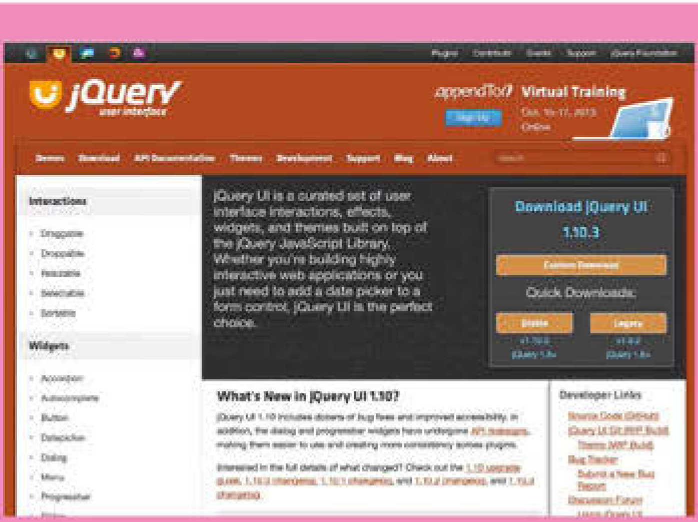
```
var el = document.querySelector('button'),
  newRoot = el.createShadowRoot();
```

Now you can place elements in your new shadow root. In this example, I'm using markup assigned to the variable `foo`:

> “Attaching JS properties to Custom Elements effectively creates a unique API for each new element”

```
var foo = '<div>Shadow DOM</div>';
newRoot.appendChild(foo);
```

The content inside `newRoot` is rendered but not available to the main DOM; so running `querySelector()` on the button element now will return a `nodeList` containing the markup of the button, not of the `div` in `foo` that was appended to the shadow root; if you apply some style rules to that `div`, there will be no effect. This is an example of encapsulation, and I'm going to



Interactions

- Draggable
- Droppable
- Resizable
- Selectable
- Sortable

Widgets

- Accordion
- Autocomplete
- Button
- Datepicker
- Dialog
- Menu
- Progressbar

jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. Whether you're building highly interactive web applications or you just need to add a date picker to a form control, jQuery UI is the perfect choice.

Download jQuery UI 1.10.3

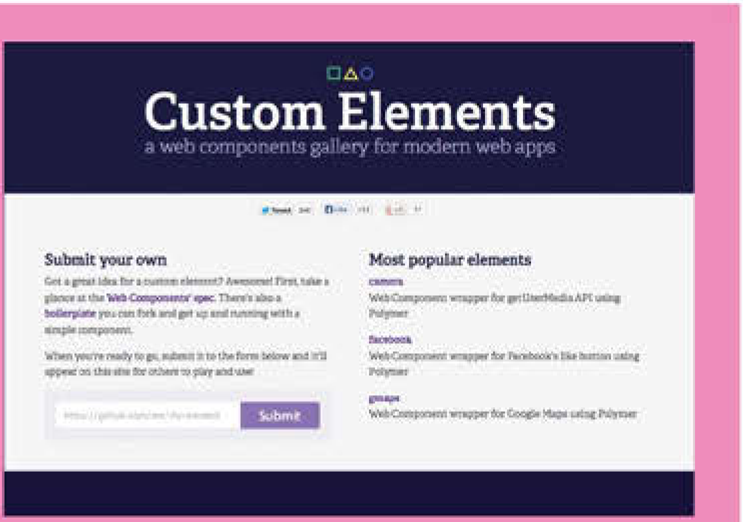
Quick Downloads:

What's New in jQuery UI 1.10?

jQuery UI 1.10 includes dozens of bug fixes and improved accessibility. In addition, the dialog and progressbar widgets have undergone API redesigns, making them easier to use and creating more consistency across plugins.

Developer Links

jQuery UI Widget libraries are prone to naming conflicts, which Web Components can prevent



Submit your own

Get a great idea for a custom element? Awesome! First, take a glance at the Web Components' spec. There's also a boilerplate you can fork and get up and running with a simple component.

When you're ready to go, submit it to the form below and it'll appear on this site for others to play and use!

Most popular elements

camera
Web Component wrapper for getUserMedia API using Polymer

facebook
Web Component wrapper for Facebook's like button using Polymer

gmaps
Web Component wrapper for Google Maps using Polymer

Good ideas Custom Elements (*customelements.io*) is a community gallery of Web Components

dig into encapsulation – and how to bypass it – in more detail shortly. But before I do that I want to move on to take a look at other parts of the Web Components suite, which are aimed at increasing the portability of code.

USING TEMPLATES

A key contributor to code reusability is using templates: blocks of code which can be repeated multiple times without having to be rewritten each time. In the past, most templating was done on the server with a language capable of building pages dynamically, such as PHP or Python.

More recently, it's become popular to use client-side templating frameworks, such as Mustache or Handlebar.js, but a drawback to this approach is that the code that's going to be applied is parsed by the browser – meaning, for example, that any linked assets (for example, the `src` of an `img` element) will be loaded.

The `template` element is here to change that. What this does in practice is render any content inside it inert, meaning it's not rendered by the browser; no linked assets will be loaded, or styles applied, or elements held in memory. You can put any markup you like inside a template, including style and script elements, and none of it will be rendered.

```
<template>
 <div>
  <h1>Hello world!</h1>
 </div>
</template>
```

In order to make use of the markup inside the template we must first select it with JS, but as it's inert we can't simply do that using `querySelector()` – it isn't rendered in the DOM, after all. Instead we must clone it using the new `content` object:

```
var tmp = document.querySelector('template'),
   foo = tmp.content.cloneNode(true);
el.appendChild(foo);
```

Once placed like this, content is active; it can be traversed normally, linked assets are downloaded, and styles applied. The `template` element combines with the Shadow DOM; you can clone content from inside a template and append it to a shadow root/s, using a single source to populate many elements:

```
newRoot.appendChild(foo);
```

This brings the ideal of having reusable components a step closer, but the next new function gets even closer still.

So far we've seen how Shadow DOM and templates make it easy to append a component to an existing element, but what if we want to create a completely bespoke element that is fully encapsulated and totally portable? Enter Custom Elements. The first step in creating a Custom Element is to choose a name. This name must contain a hyphen to distinguish it from existing (and future) elements. Once the name is chosen, you pass it into the new `register` interface to create the element:

```
document.register('foo-bar');
```

The long-term plan is to also allow elements to be created using declarative HTML, via the `element` element, but this is not fully scoped yet. Anyway, at this point the element has been registered and is ready for use in your markup:

```
<foo-bar></foo-bar>
```

But while it's a new element, it doesn't really do anything special yet. The real utility in Custom Elements comes in

▶

▶ adding bespoke functionality to them through the content inside. Like the `template` element, you can put any markup inside a Custom Element – the difference being it will be parsed by the document, unless attached inside a Shadow DOM.

EXTRA-USEFUL ELEMENTS

Custom Elements become extra-useful when you attach their own JS properties and methods to them – in essence, creating a unique API for each new element. You do this by creating the new element with its own prototype:

```
var FooBarProto = Object.create(HTMLElement.prototype),
   FooBar = document.register('foo-bar',{prototype: FooBar-
   Proto});
```

If you'd prefer to extend existing elements rather than create your own, you can do so by inheriting the prototype of an existing element type – in the example overleaf, I'm going to extend a button:

```
var FooBtnProto = Object. create(HTMLButtonElement.proto-
type),
   FooBtn = document.register('foo-btn',{prototype: FooBtnPro-
   to});
```

Next, associate the name of your Custom Element to the element it's extending through use of the `is` attribute:

```
<button is="foo-btn">Hello, world!</button>
```

With Custom Elements, not only do we get the portability we've been seeking but can also name elements after their function. For example, a button which renders a calendar could be called `button-calendar` – markup becomes meaningful again!

HTML IMPORTS

So how can we make components themselves more easily shareable? Obviously if you're using JS to create components, you can link to an external JS file, but that isn't so easy to do for HTML. At least, it wasn't until now. HTML imports have been created to perform exactly this function. Using the `link` element

Essential viewing The Polymer library project is key to the development of Web Components

THE POLYMER PROJECT

Development of the different features that make up Web Components is being handled in very unique way, through the Polymer project (**polymer-project.org**). Polymer is a new type of library, both 'forward polyfilling' all parts of the proposed Web Components suite, and building a web application framework on top of it.

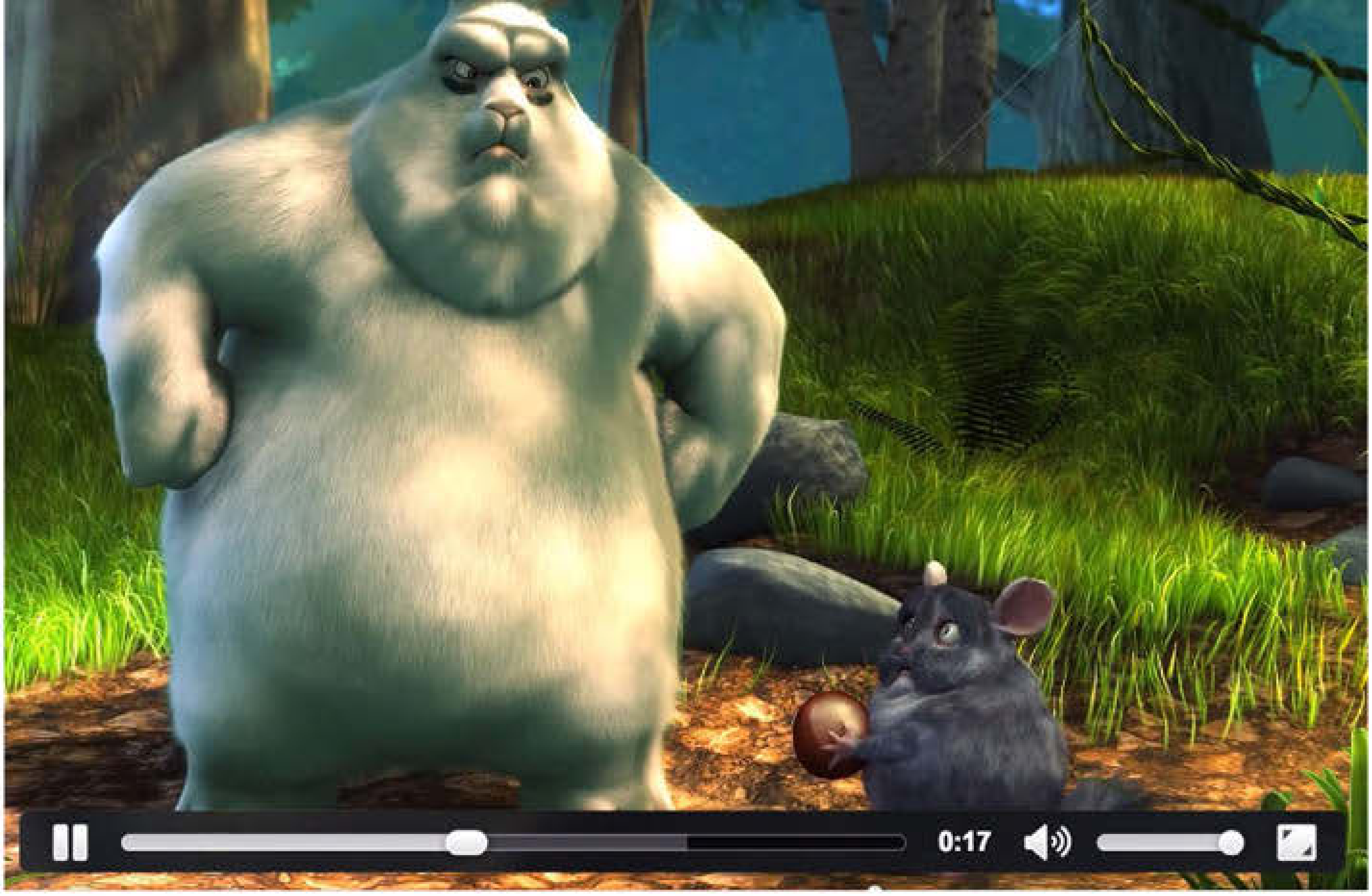
Polymer provides a development environment that's mutually beneficial to spec authors and developers: authors are able to test and alter properties without worrying that partial or experimental implementations will be used by developers, while also getting feedback on what works and what doesn't; and devs can test the latest Web Components features without waiting for browser implementation.

The fruit of this mutual development is that, when the spec is stable, there will be a ready-made polyfill library allowing Web Components to be implemented backwards-compatibly. This approach has coined the word 'prollyfill', for a library that implements non-standard features to aid in their development until eventually becoming a polyfill for standard features.

The Polymer approach looks set to become the template for web platform development in the future. It's a completely open way of working which involves all key parties from the very beginning, but without running into the legacy problems caused by the previous approach: using prefixed properties.

To try Polymer for yourself, just download the latest build and load `polymer.min.js` in your document. Once done, you have complete access to all of the key web component features; natively if they're implemented in your browser, and polyfilled if they're not.

With new technologies, the temptation is to think, 'I'll wait to learn about this when it's more widely implemented', but Web Components are without doubt the future of frontend development, and Polymer exists to let you start learning about them immediately.



Back door Video elements – like this one for the animated short *Big Buck Bunny* – have controls made from web platform technologies. However, they aren't accessible without the Shadow DOM

with a value of `import` for the `rel` attribute you can include external files into your document.

```
<link rel="import" href="foo.html">
```

Note that these won't be parsed into the DOM: they'll just be loaded into the browser memory and available for you to access through JS. In the code that follows, I'm selecting my previously defined Custom Element, `foo-bar`, and registering it into the DOM of my current document:

```
var link = document.querySelector('link[rel=import]'),
   foo = link.import.querySelector('foo-bar');
document.register('foo-bar');
el.appendChild(foo);
```

HTML imports are available in Chrome but face a slightly uncertain future, as the Firefox team has announced that due to some weaknesses in the specification, and possible alternative methods, it won't be implementing them in its browser.

ENCAPSULATION AND SCOPE

In this article, I've talked about defining your own elements and reusing code, but perhaps haven't clearly explained how the encapsulation provided by the Shadow DOM works. There is a boundary between the content inside the shadow root and the content outside it; this is known, inevitably, as the 'shadow boundary'. That isn't to say that the content inside the shadow root can't be accessed – just that it can't be accessed through regular means of selection, without special permission.

For example, using JS you won't be able to access any part of the Shadow DOM using `querySelector()`, `getElementById()`, or any similar method, unless you first use the `shadowRoot` attribute to return an object containing the nodes in the shadow root:

```
var el = document.querySelector('button'),
   rootNodes = el.shadowRoot,
   rootEls = rootNodes.querySelector('div');
```

Likewise, CSS rules from the current document won't be applied to elements inside any shadow DOMs on your page without using special selectors. For example, you can use the `::shadow` pseudo-element to select the root of the shadow tree – so to apply rules to element #bar inside the shadow DOM of element `# foo` you could write this selector:

```
#foo::shadow #bar { ... }
```

There are several other style rules being drawn up in the CSS Scoping Module, which is still in the early stages of the editing process.

IN CONCLUSION

In my opinion, Web Components are the biggest change to web development in the last 10 years – bigger even than HTML5. All of the major browser vendors are working on implementations (at least partially), so even if the details end up being slightly different to the ones I've outlined here, this is the future.

Turn to page 86 for Rob Dodson's tutorial on how to build an app with the Polymer library

The Pro's Guide to

Responsive Web Design

Advanced techniques to build better sites for any device

AUTHOR

JUSTIN AVERY

By day, Justin (@justinavery) is a technical consultant, and by night he curates the RWD Weekly newsletter, hosts a responsive podcast and runs *responsivedesign.is*

ILLUSTRATION

MARTINA FLOR

Berlin-based Martina creates type, lettering and illustration work for clients all over the world *martinaflor.com*

Responsive web design sounds incredibly simple. Opt for flexible grids for the layout, use flexible media (images, video, `iframes`), and apply media queries to update these measurements to best arrange content on any viewport. In practice we've learnt it is not really that easy. Tiny issues that crop up during every project keep us scratching our heads, and occasionally even carving fingernail trenches on our desks.

Since I began curating the Responsive Design Weekly newsletter (*responsivedesignweekly.com*) I've been fortunate enough to speak with many members of the web community and hear their experiences. I wanted to find out exactly what the community really wanted to learn, so I circulated a survey to readers. Here are the top results:

1. Responsive images
2. Improving performance
3. Responsive typography
4. Media queries in JavaScript
5. Progressive Enhancement
6. Layout

With those topics in mind, I ran a series of podcasts (*responsivedesign.is/feeds/podcast*) asking some of our industry leaders for their thoughts. In their responses, one point was unanimous: focus upon getting the basics right before you start worrying about exciting and advanced techniques. By taking things back to the basics, we are able to build a robust interface for everyone, layering on features when the device or user's context allows.

"So … what about these advanced techniques?" I hear you ask. I think Stephen Hay summed it up best when he said: "The ultimate RWD technique is to start off by not using any advanced RWD techniques. Start with structured content and build your way up. Only add a breakpoint when the design breaks and the content dictates it and … that's it."

In this article, I'll begin with the basics and add layers of complexity as the situation allows, to build up to those advanced techniques. Let's get started.

RESPONSIVE IMAGES

Fluid media was a key part of RWD when it was first defined by Ethan Marcotte. `width: 100%; , max-width: 100%;` still works today, but the responsive image landscape has become a lot more complicated. With increasing numbers of screen sizes, pixel density and devices to support we crave greater control.

The three main concerns were defined by the Responsive Images Community Group (RICG):

1. The kilobyte size of the image we are sending over the wire
2. The physical size of the image we are sending to high DPI devices
3. The image crop in the form of art direction for the particular size of the viewport

Yoav Weiss, with help from Indiegogo (*netm.ag/blink-255*), has done the majority of the work on the Blink implementation – Google's fork of WebKit, and by the time you're reading this it will be shipped in Chrome and Firefox. Safari 8 will ship `srcset`, however the `sizes` attribute is only in nightly builds and `<picture>` is not yet implemented.

With the arrival of anything new to our web development process, it can be difficult to get started. Let's run through an example step by step.

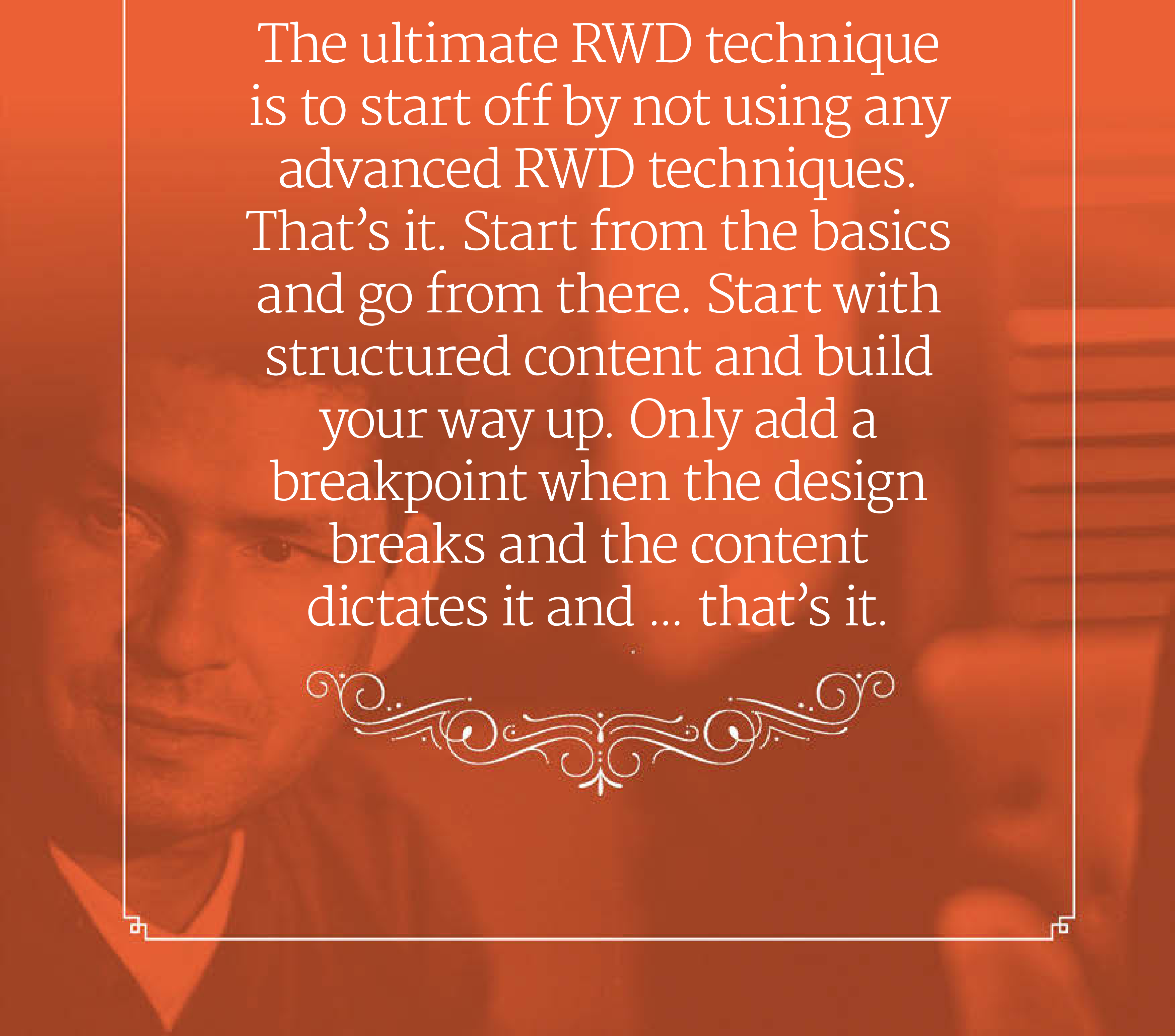
```
 <img
   <!-- Declare the fallback image for all non picture supporting browsers -->
   src="horse-350.jpg"
   <!-- Declare all of the image sizes in srcset. Include the image width using the w descriptor to inform the browser of the width of each image.-->
   srcset="horse-350.jpg 350w,
   horse-500.jpg 500w,
   horse-1024.jpg 1024w,
   horse.jpg 2000w"
   <!-- Sizes inform the browser of our site layout. Here we're saying for any viewport that is 64ems and bigger, use an image that will occupy 70% of the viewport -->
   sizes="(min-width: 64em) 70vw,
   <!-- If the viewport isn't that big, then for any viewport that is 37.5ems and bigger, use an image that occupies 95% of the viewport -->
   (min-width: 37.5em) 95vw,
   <!-- and if the viewport is smaller than that, then use an image that occupies 100% of the viewport-->
   100vw"
   <!-- always have alt text.-->
   alt="A horse" />
```

From a performance point of view it doesn't matter if you use `min-width` or `max-width` in the `sizes` attribute – but the source order does matter. The browser will always use the first matching size.

Also, remember we are hard-coding the `sizes` attribute to be directly defined against our design.

▶

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL



Stephen Hay

The ultimate RWD technique is to start off by not using any advanced RWD techniques. That’s it. Start from the basics and go from there. Start with structured content and build your way up. Only add a breakpoint when the design breaks and the content dictates it and … that’s it.

Cutting the mustard

'Cutting the mustard' is an approach that involves testing for specific device features. If these features exist then the user is served a more advanced and immersive experience, while those that do not support them will still get the content they were looking for.

The test effectively separates users into two groups; those with HTML5 capabilities and those without. The same initial content is delivered to both sets, however the HTML5 group will also benefit from `localStorage`, `querySelector` and `addEventListener`.

```
if('querySelector' in document
    && 'localStorage' in window
    && 'addEventListener' in window) {
    // bootstrap the javascript application
    }
```

Cutting the mustard allows us to organise users into tiers based on feature detection. These tiers all receive a different website experience, but every user still has access to the basic content.

There's a useful BBC article at *netm.ag/mustardarticle-260*, or check out a presentation by the BBC's Tom Maslen – who first coined the term 'cutting the mustard' – at *netm.ag/mustardvid-260*.

This may cause issues moving forwards. For example, if you redesign your site, you'll need to revisit all of the `<img>` or `<picture>`s and redefine the sizes. If you are using a CMS, this can be overcome as part of your build process.

WordPress already has a plugin to help (*netm.ag/picture-260*). It defines the `srcset` on WP standard image varieties and allows you to declare `sizes` in a central location. When the page is generated from the database, it swaps out any mentions on `<img>` and replaces them with the `picture` markup.

Basic

- Think about whether you really need to include an image. Is the image core content, or is it decorative? One less image will mean a faster load time
- Optimise the images you do need to include using ImageOptim (*imageoptim.com*)
- Set expire headers for your images on your server or .htaccess file (see details under 'Performance')
- PictureFill (*github.com/scottjehl/picturefill*) provides polyfill support for pictures

Advanced

- Lazy load your images using jQuery's Lazy Load plugin (*appelsiini.net/projects/lazyload*)
- Use `HTMLImageElement.Sizes` and `HTMLPictureElement` for feature detection.
- The advanced PictureFill WP plugin, found on Github (*netm.ag/imgtags-260*), will allow you to define custom image widths and sizes values

PERFORMANCE

To get the fastest perceived performance on our pages, we need all the HTML and CSS required to render the top part of our page within the first response from the server. That magic number is 14kb and is based on the max congestion window size within the first round-trip time (RTT). Patrick Hamann, frontend technical lead at the *Guardian* (*theguardian.com*), and his team have managed to break the 1000ms barrier using a mixture of frontend and backend techniques. The *Guardian*'s approach is to ensure the required content – the article – is delivered to the user as quickly as possible and within the 14kb magic number.

Let's look at the process:

1. User clicks on a Google link to a news story
2. A single blocking request is sent to the database for the article. No related stories or comments are requested
3. The HTML is loaded containing Critical CSS in the `<head>`
4. A 'Cut the mustard' process is undertaken and any conditional elements based upon the user's device features are loaded
5. Any content related to or supporting the article itself (related article images, article comments and so on) are lazy loaded into place

Optimising the critical rendering path like this means the `<head>` is 222 lines long. However, the critical content the user came to see still comes within the 14kb initial payload when gzipped. It's this process that helps break that 1000ms rendering barrier.

Conditional and lazy loading

Conditional loading improves the user's experience based on their device feature. Tools like Modernizr allow you to test for these features, but be aware that just because a browser says it offers support, that doesn't always mean full support.

One technique is to hold off loading something until the user shows intent to use that feature. This would be considered conditional. You can hold off loading in the social icons until the user hovers over or touches the icons, or you could avoid loading an `iframe` Google Map on smaller viewports where the user is likely to prefer linking to a dedicated mapping application. Another approach is to 'Cut the mustard' – see boxout above for details on this.

Lazy loading is defined as something that you always intend to load on the page – images that are a part of the article, comments or other related articles – but that don't need to be there for the user to start consuming the content.

Basic

- Enable gzipping for files and set expire headers for all static content (*netm.ag/expire-260*)
- Use the Lazy Load jQuery plugin (*appelsiini.net/projects/lazyload*). This loads images when approaching the viewport, or after a certain period of time

Advanced

- Set up Fastly (*fastly.com*) or CloudFlare (*cloudflare.com*). Content delivery networks (CDNs) deliver your static content to users faster than your own server, and have some free tiers
- Enable SPDY for http2-enabled browsers to take advantage of http2 features like parallel http requests
- Social Count (*filamentgroup.com/lab/socialcount*) allows for conditional loading of your social icons
- Using the Static Maps API will allow you to switch out Interactive Google maps for images. Take a look at Brad Frost's example at *netm.ag/static-260*
- Ajax Include Pattern (*netm.ag/ajax-260*) will load content snippets from either a `data-before`, `data-after` or `data-replace` attribute

RESPONSIVE TYPOGRAPHY

Typography is about making your content easy to digest. Responsive typography extends this to ensure readability across a wide variety of devices and viewports. Jordan Moore admits that type is one thing he isn't willing to budge on. Drop an image or two if you need, but make sure you have great type.

Stephen Hay suggests setting the HTML font size to 100 per cent (read: just leave it as it is) because each browser or device manufacturer makes a reasonably readable default for a particular resolution or device. For most desktop browsers this is 16px.

On the other hand, Moore uses the REM unit and HTML `font-size` to set a minimum font size for certain content elements. For example, if you want the byline of an article to always be 14px, then set that as the base `font-size` on the HTML element and set `.byline { font-size: 1rem;}`. As you scale the `body: font-size:` to suit the viewport you will not impact the `.by-line` style.

A good reading line length is also important – aim for 45 to 65 characters. There's a bookmarklet you can use to check your content (*netm.ag/bookmarklet-260*). If you are supporting multiple languages with your design, then line length may vary as well. Moore suggests using `:lang(de) article {max-width: 30em}` to cover off any issues there.

To maintain vertical rhythm, set `margin-bottom` against content blocks, `<ul>`, `<ol>`, `<blockquote>`, `<table>`, `<blockquote>` and so on, to the same as your `line-height`. If the rhythm is interrupted with the introduction of images you could fix it by adding Baseline.js (*netm.ag/baseline-260*) or BaselineAlign.js (*netm.ag/align-260*).

Basic

- Base your font on 100 per cent body
- Work in relative em units
- Set your margins to your line height to maintain vertical rhythm in your design

Advanced

- Improve font loading performance with Enhance.js (*github.com/filamentgroup/enhance*) or deferred font loading (*netm.ag/defer-260*)
- Use Sass `@includes` for semantic headings. Often we need to use the `h5` style in a sidebar widget that requires `h2` markup. Use Bearded's Typographic Mixins (*netm.ag/mixin-260*) to control the size and remain semantic with the below code:

```
.sidebar h2 {
  @include heading-5
 }
```

MEDIA QUERIES IN JAVASCRIPT

Ever since we have been able to control the layout across a variety of viewports through media queries, we've been looking for a way to tie that into running our JavaScript as well. There are a few ways to fire JavaScript on certain viewport widths, heights and orientations, and some smart people have written some easy-to-use native JS plugins like Enquire.js (*netm.ag/enquire-241*) and Simple State Manager (*netm.ag/simple-260*). You could even build this yourself using `matchMedia`. However, you have the issue that you need to duplicate your media queries in your CSS and JavaScript.

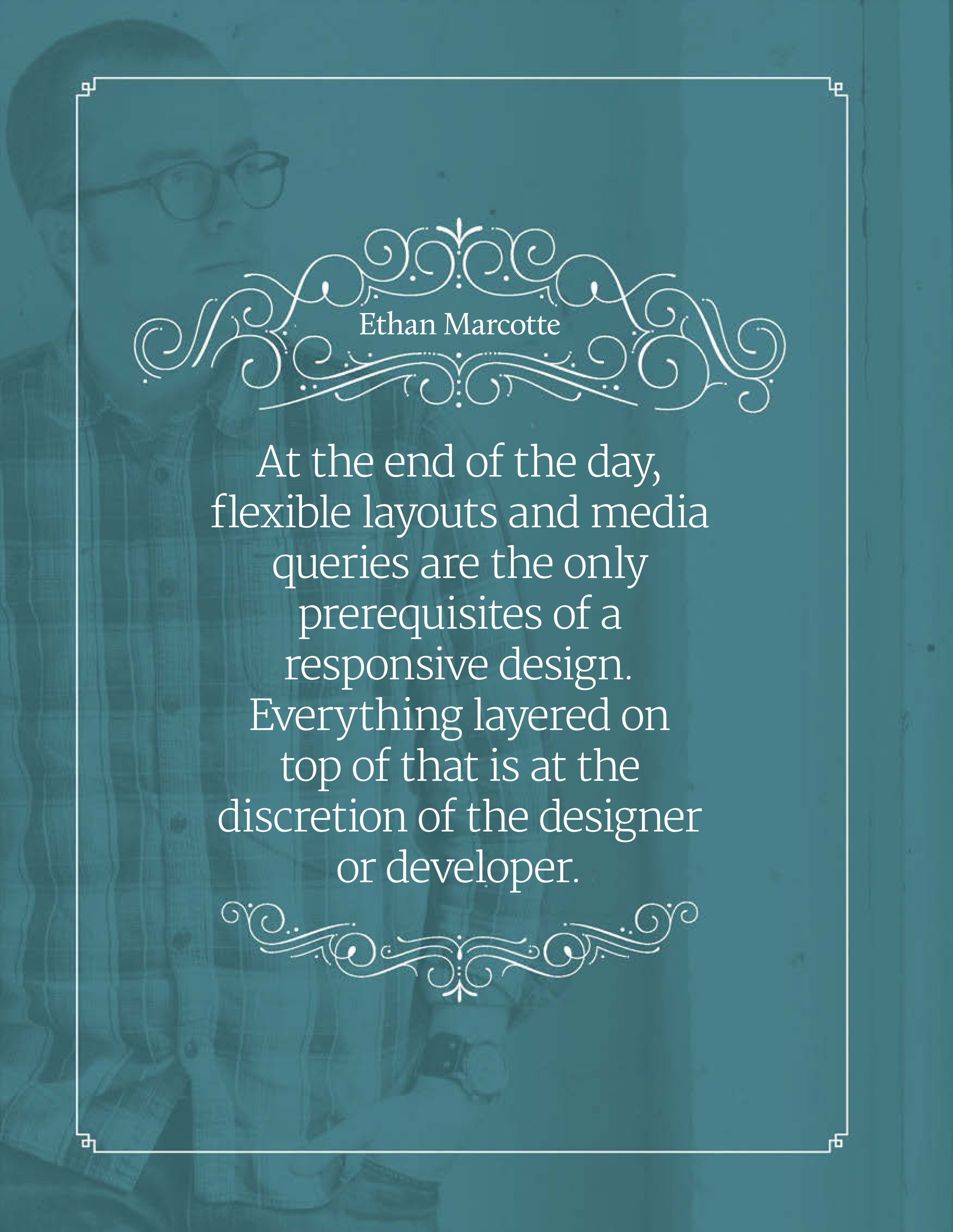
Responsibly responsive

Web designer **Scott Jehl** asks if RWD is bad for performance

I think the web in general has a problem with bloated pages, and that problem both preceded and continues alongside responsive sites. In that light, I don't think it's fair to attribute the performance issues found in many responsive sites to RWD itself, but rather to a broader lack of follow-through with widely-recommended optimisations. RWD is defined as nothing more than a means for adapting visual layout, and it's the best approach we have for doing that.

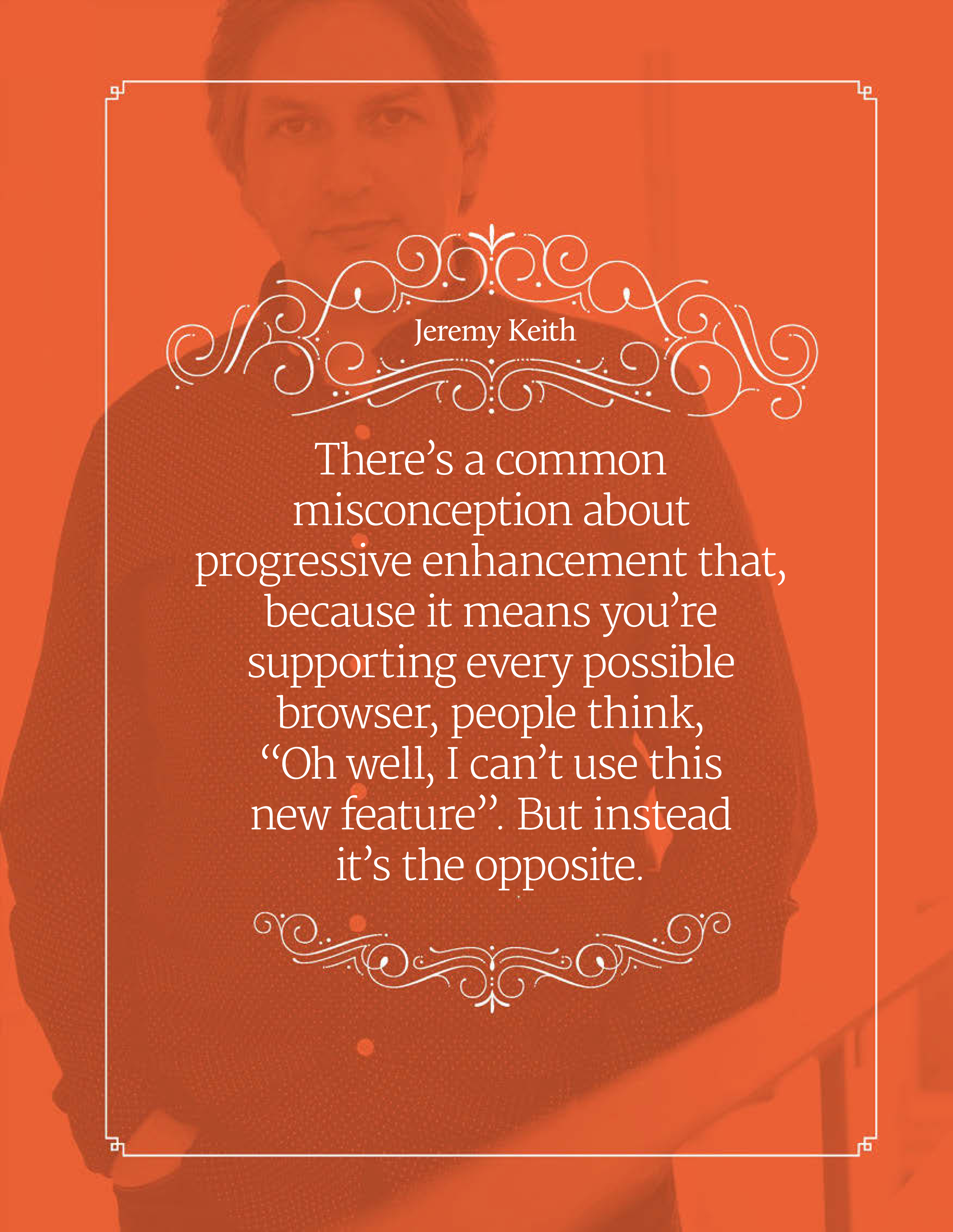
All of the performance recommendations we need to follow for building sites in general – concatenating, minifying or gzipping code, optimising imagery, caching and so on – should be applied to responsive sites as well. Sometimes it's tricky to pull off with multi-device code, but I think it's more of a challenge than an impossibility.

Responsible Responsive Design by Scott Jehl is available to purchase at netm.ag/responsible-260



Ethan Marcotte

At the end of the day, flexible layouts and media queries are the only prerequisites of a responsive design. Everything layered on top of that is at the discretion of the designer or developer.



Jeremy Keith

There's a common misconception about progressive enhancement that, because it means you're supporting every possible browser, people think, "Oh well, I can't use this new feature". But instead it's the opposite.

▶ Aaron Gustafson has a neat trick that means you don't have to manage and match your media queries in your CSS and your JS. The idea originally came from Jeremy Keith (*adactio.com/journal/5429*) and in this example Gustafson has taken it to a full implementation.

Using `getActiveMQ` (*netm.ag/media-260*), inject `div#getActiveMQ-watcher` at the end of the body element and hide it. Then within the CSS set `#getActiveMQ-watcher {font-family: break-0;}` to the first media query, `font-family: break-1;` to the second, `font-family: break-2;` to the third and so on.

The script uses `watchResize()` (*netm.ag/resize-260*) to check to see if the size of the viewport has changed, and then reads back the active `font-family`. Now you can use this to hook JS enhancements like adding a tabbed interface to a `<dl>` when the viewport allows, changing the behaviour of a lightbox, or updating the layout of a data table. Check out the `getActiveMQ` Codepen at *netm.ag/active-260*.

Basic

➤ Forget about JavaScript for different viewports. Provide content and website functions to users in a way they can access it across all viewports. We should never *need* JavaScript

Advanced

➤ Extend Gustafson's method by using Breakup (*github.com/BPScott/breakup*) as a predefined list of media queries and automating the creation of the list of font families for `getActiveMQ-watcher`

PROGRESSIVE ENHANCEMENT

A common misconception about progressive enhancement is that people think, "Oh well I can't use this new feature", but really, it's the opposite. Progressive enhancement means that you can deliver a feature if it's only supported in one or even no browsers, and over time people will get a better experience as new versions arrive.

If you look at the core function of any website, you can deliver that as HTML and have the server side do all the processing. Payments, forms, Likes, sharing, emails, dashboards – it can all be done. Once the basic task is built we can then layer the awesome technologies on top of that, because we have a safety net to catch those that fall through. Most of the advanced approaches we have talked about here are based upon progressive enhancement.

LAYOUT

Flexible layout is simple to say, but it has many different approaches. Create simple grid layouts with less markup by using `:nth-child()` selector.

```
 /* declare the mobile first width for the grid */
 .grid-1-4 { float:left; width: 100%; }

 /* When the viewport is at least 37.5em then set the
grid to 50% per element */
 @media (min-width: 37.5em) {
  .grid-1-4 { width: 50%; }

 /* Clear the float every second element AFTER the
first. This targets the 3rd, 5th, 7th, 9th... in the grid.*/
  .grid-1-4:nth-of-type(2n+1) { clear: left; }
 }

 @media (min-width: 64em) { .grid-1-4 { width: 25%;
}
/* Remove the previous clear*/
  .grid-1-4:nth-of-type(2n+1) { clear: none; }
/* Clear the float every 4th element AFTER the first.
This targets the 5th, 9th... in the grid.*/
  .grid-1-4:nth-of-type(4n+1) { clear: left; }
 }
```

Wave goodbye to using `position` and `float` for your layouts. While they have served us well to date, their use for layout has been a necessary hack. We've now got two new kids on the block that will help fix all our layout woes – Flexbox and Grids.

Flexbox is great for individual modules, controlling the layout of pieces of content within each of the modules. There are layouts we attempt to deliver that can be more easily achieved using Flexbox, and this is even more true with responsive sites. For more on this, check out CSS Tricks' guide to Flexbox (*netm.ag/flexbox1-26*) or the Flexbox Polyfill (*flexiejs.com*).

CSS grid layout

Grid is more for the macro level layout. The Grid layout module gives you a great way to describe your layout within your CSS. While it's still in the draft stage at the moment, I recommend this article on the CSS Grid layout by Rachel Andrew (*netm.ag/grid-260*).

FINALLY

These are just a few tips to extend your responsive practice. When approaching any new responsive work, take a step back and ensure that you get the basics right. Start with your content, HTML and layer improved experiences upon them and there won't be any limit to where you can take your designs. ◘

Thanks

With thanks to Jeremy Keith, John Allsopp, Stephen Hay, Aaron Gustafson, Ethan Marcotte, Brad Frost, Patrick Hamann, Jordan Moore and Scott Jehl, who all helped form the thoughts in this article.

Resources

PERFORMANCE

Network Link throttling (*netm.ag/stuntbox-260*)

Chrome Dev Tools: Canary (*netm.ag/canary-260*)

Enhance.js (*netm.ag/enhance-260*)

Github South Street (*netm.ag/southstreet-260*)

POLYFILLS

Respond.js (*netm.ag/respond-260*)

Picture.js (*netm.ag/fill-241*)

Flexbox Polyfill (*flexiejs.com*)

PLUGINS

TableSaw (*netm.ag/tablesaw-260*)

Fit Vids (*fitvidsjs.com*)

Fit Text (*fittextjs.com*)

Adaptive UIs (*netm.ag/allsopp-260*)

Label Mask (*netm.ag/label-260*)

ALL THINGS RWD

Responsive Design Weekly (*netm.ag/newsletter-260*)

This is Responsive (*netm.ag/thisis-260*)

Responsive Design.is (*responsivedesign.is*)

SUBSCRIBE TO NET

GET THE NO.1 CHOICE FOR WEB DESIGNERS AND DEVELOPERS DELIVERED TO YOUR DOOR, YOUR DEVICE, OR BOTH

50 MINUTES OF FREE VIDEO TUTORIALS
The voice of web design
net
THE TOP 20 WEB DESIGN TRENDS 2015
Must-know techniques to build your best ever site
BUILD THE PERFECT WEBSITE

PRINT EDITION ONLY

Take out a print subscription to **net** and get your copy before it hits the shops. Each issue is packed with the latest web trends, technologies and techniques

FROM

£26.99

SAVE UP TO 38%

BASED ON A 6-MONTH SUBSCRIPTION

The voice of web design
net
RESPONSIVE SITES WIT FOUNDAT
CLIMB TO THE TOP OF GOOGLE
net
DISCOVER THE POWER OF SVG EFFECTS
BUILD MOBILE APPS THE EASY WAY

DIGITAL EDITION ONLY

Take out a digital subscription to **net** for on-the-go access to our fully interactive edition, with streaming screencasts, extra images and more

FROM

£20.49

SAVE UP TO 45%

BASED ON A 6-MONTH SUBSCRIPTION

Terms & conditions: Prices and savings quoted are compared to buying full priced UK print and digital issues. You will receive 13 issues in a year. If you are dissatisfied in any way you can write to us or call us to cancel your subscription at any time and we will refund you for all un-mailed issues. Prices correct at point of print and subject to change. For full terms and conditions please visit: *myfavm.ag/magterms*. Offer ends 31 May 2015.

SPECIAL 20TH ANNIVERSARY ISSUE
The voice of web design
net
FLEXIBLE WORDPRESS LAYOUTS
HANDLEBARS
PROTOTYPE WITH ORIGAMI
ESSENTIAL Javascript
GRUNT
Gulp
BACKBONE
YEOMAN
ember.js
ANGULAR
PRO IMAGE OPTIMISATION TECHNIQUES
TOP 10 WEB DESIGN TOOLS For 2015
We review the hottest design tools for 2015
BUILD NATIVE APPS WITH HTML & CSS
FREE DIGITAL EDITION!
INTRODUCING WORDPRESS 4.0

NEW PRINT & DIGITAL EDITION

Enjoy a combined print and digital subscription, and take advantage of print as well as exploring the fully interactive digital experience

GREAT REASONS TO SUBSCRIBE

- Print edition delivered to your door
- 13 issues in a one-year subscription
- iPad and iPhone edition download
- Android edition download
- Money-back guarantee

FROM
£32.49

SAVE UP TO 60%

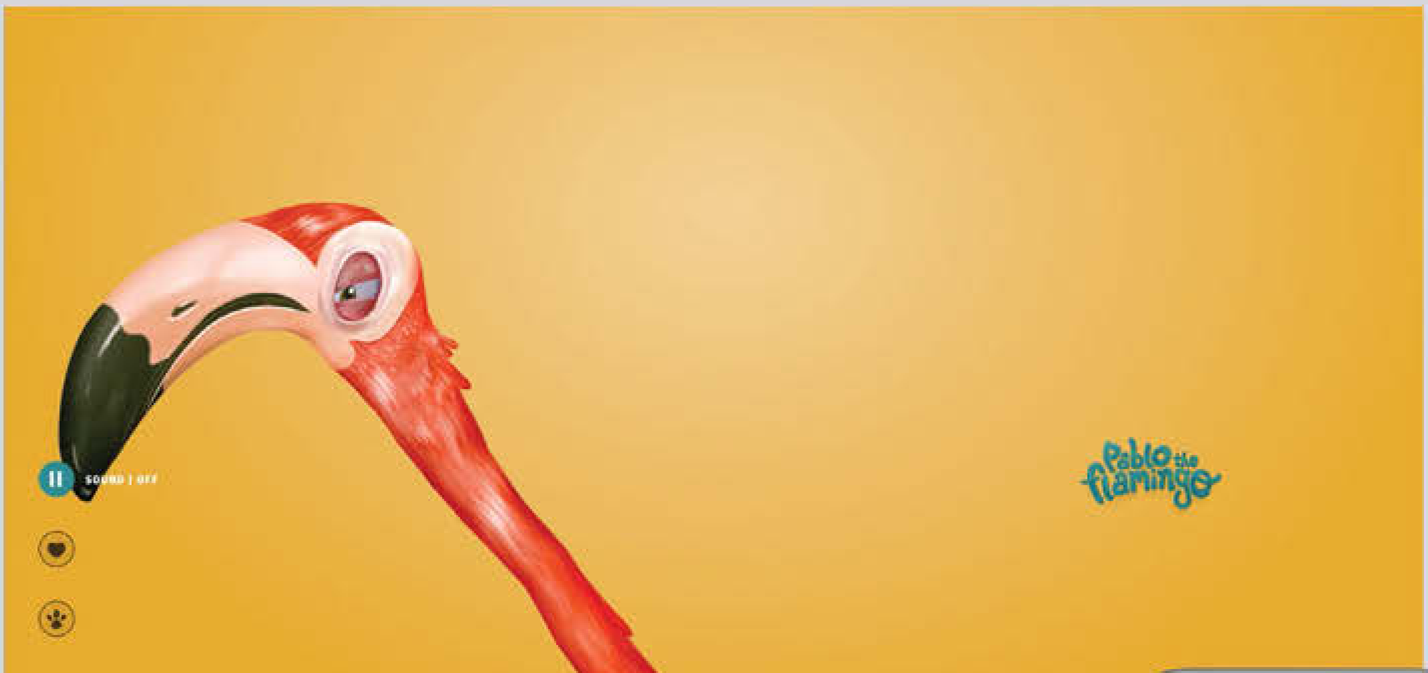
TWO SIMPLE WAYS TO SUBSCRIBE

ONLINE: myfavouritemagazines.co.uk/NETSUBS
PHONE: 0844 848 2852 AND QUOTE CODE:
PRINT15, DIGITAL15 OR BUNDLE15

Your subscription will then continue at £32.49 every 6 months - saving 31% on the shop price and giving you an 83% discount on a digital subscription

GALLERY

Sensational websites selected by our panel of experts



★ THREE.JS, MATTER.JS, AUTODESK MAYA, WEBGL, WEB AUDIO

PABLOTHEFLAMINGO.COM

Nathan Gordon *cinemont.com*, **Pascal van der Haar** *netm.ag/haar-262*, **Jono Yuen** *jonoyuen.com*

A one-trick site, but what a delightful trick! Pablo is a flamingo who loves to dance, and he's brought to life beautifully in fully responsive WebGL. Pablo is the result of a collaboration between authors Nathan Gordon, Pascal van der Haar and Jono Yuen.

It isn't a promotional site, but a side-project that came about through a conversation between web developer Gordon and art director Van der Haar. Pablo began life as a model constructed in Autodesk Maya. This was then exported as a simple grid mesh, to which the illustrations by designer Yuen were added as a texture. The models were then brought into the browser, where 3D modelling library three.js and physics library Matter.js were used to bring Pablo to life online.

"I was able to create a precise form that could be directly mapped to the physics base. This gave me the result that I was after, and was the largest 'aha!' moment of the whole project," says Gordon.

The response from the public has been incredible – in his first week online Pablo was seen 60,000 times, won FWA Site of the Day and earned countless plaudits across social media.

PETER GASSTON

Peter is a veteran web developer who now works as a technologist and frontend lead at rehabstudio. He's the author of *The Book of CSS3* and *The Modern Web*
w: *about.me/petergasston*
t: @stopsatgreen

KAROLINA SZCZUR

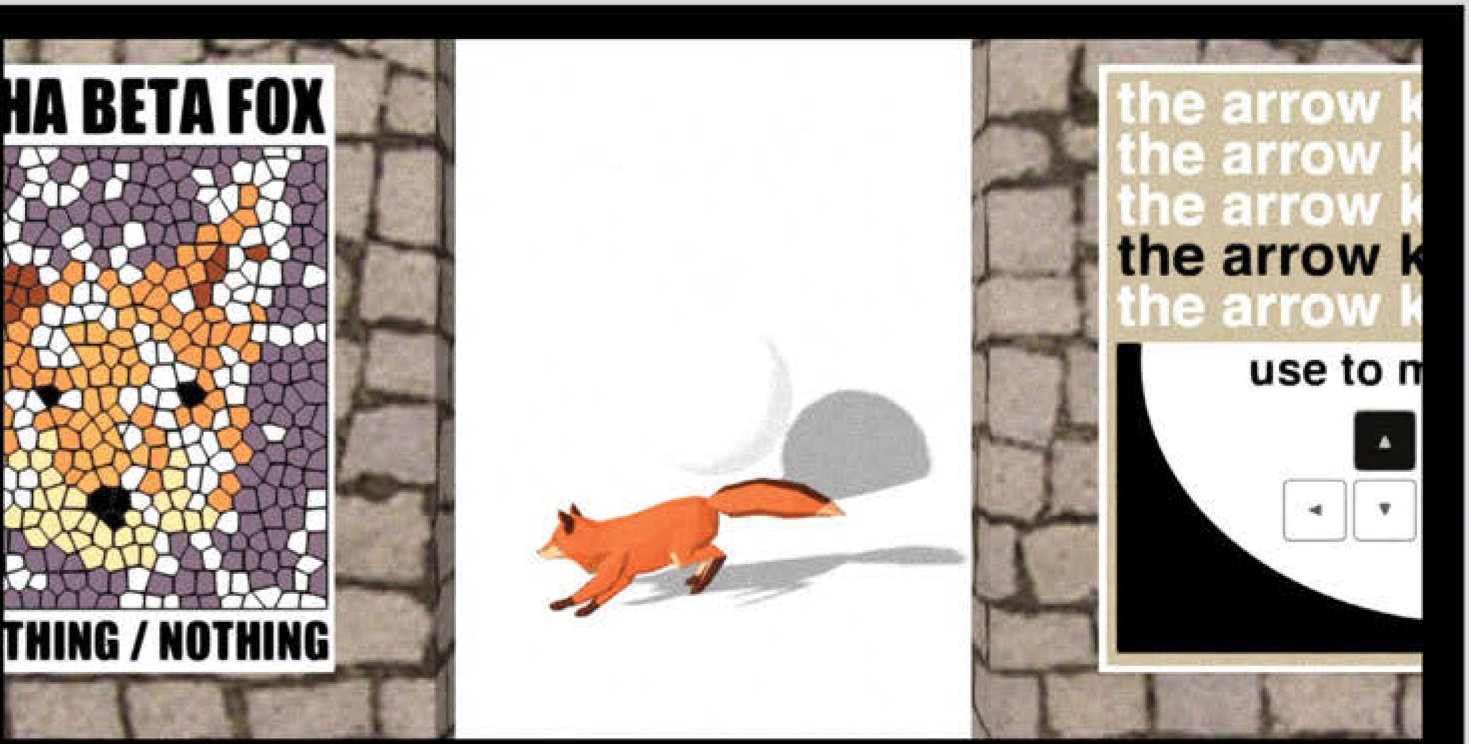
Karolina is a designer, developer, photographer and writer at &yet. She's also an open source aficionado and runs CSSConf in Oakland, California
w: *thefox.is*
t: @fox

DAN EDWARDS

Dan is a freelance web and user interface designer, and one half of oozled (*oozled.com*) – an online directory of creative design and development resources
w: *danedwards.me*
t: @de

GERI COADY

Geri is a colour-obsessed freelance illustrator and web designer, author of *A Pocket Guide to Colour Accessibility* and **net**'s Designer of the Year 2014
w: *hellogeri.com*
t: @hellogeri



THREE.JS, WEB AUDIO API, WEBGL

SOMETHINGNOTHING.NET

James Reid for Alpha Beta Fox *facebook.com/AlphaBetaFox*

Alpha Beta Fox is a self-described 'dreamy shoegaze' band from Adelaide, Australia, and *SomethingNothing.net* is an interactive music video for its song 'Something / Nothing'. The futuristic, psychedelic video fits the music style perfectly.

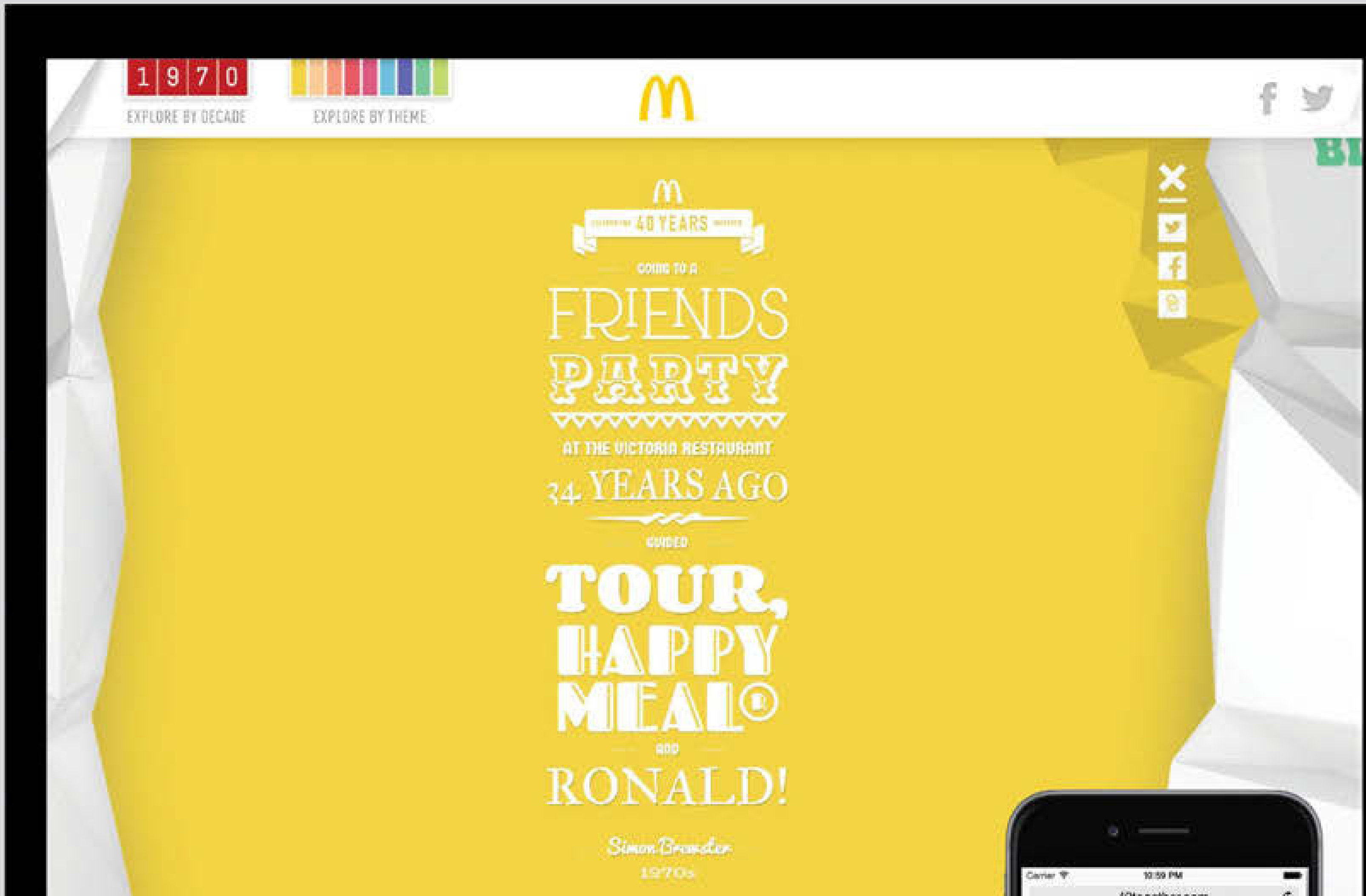
Built by Adelaide-based developer and friend of the band James Reid, the site uses the three.js WebGL library for its 3D graphics, and the Web Audio API to play the music. At one point, viewers even get the chance to alter a video clip of the band by changing parameters with dat.GUI. It seems the site was something of a learning experience for Reid, as he included many references to classic processing algorithms and models from other 3D graphics projects: Game of Life cellular automaton, the 4D tesseract cube, Kaliset fractals, Hopalong Orbits ...

"James was reading about tesseracts, then thought, 'Why not put it in the video, with people dancing around it in space?'" Alpha Beta Fox singer Tanya Giobbi says. "We thought it was pretty awesome."

The site isn't optimised for mobile, but it does work in one, less common environment: it has a VR mode for Oculus Rift or similar devices.

> "A hugely enjoyable interactive WebGL music video experience. Desktop-only though"
>
> PAUL IRISH (@PAUL_IRISH)

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

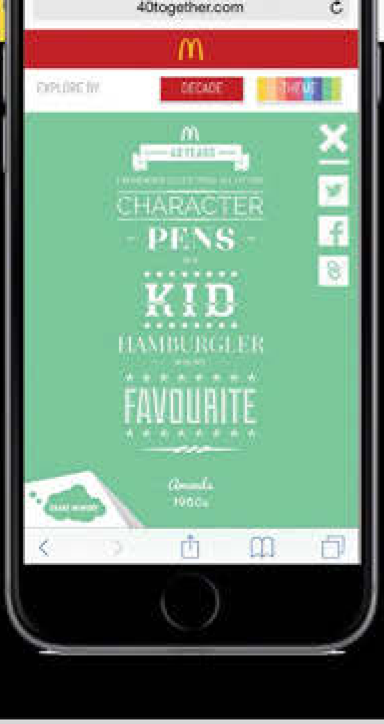


1970
EXPLORE BY DECADE
EXPLORE BY THEME

40 YEARS

GOING TO A
FRIENDS
PARTY
AT THE VICTORIA RESTAURANT
34 YEARS AGO
GUIDED
TOUR,
HAPPY
MEAL®
AND
RONALD!

Simon Brewster
1970s



Carrier
10:59 PM
40together.com

EXPLORE BY
DECADE
THEME

40 YEARS
CHARACTER
PENS
KID
HAMBURGLER
FAVOURITE

Amanda
1960s

★ HTML5 CANVAS API, PIXI.JS WEBGL, WEB SPEECH API

40TOGETHER.COM

Razorfish *razorfish.com*, **Goodboy Digital** *goodboydigital.com*

This year marks McDonald's' 40th anniversary, and it's celebrating by presenting customers' memories spanning five decades, in an interactive application created by Razorfish. Built entirely using the HTML5 Canvas API, there are two things here that really catch the eye: the beautifully animated interface, and good-enough-to-eat typography.

The grid layout and posters were built using Pixi.js, a 2D WebGL renderer by Goodboy Digital. A custom 3D renderer was built on top to provide the transitional animations and make them perform smoothly, even on mobile devices. The system that creates the typographic posters uses natural language and syntax analysis to deconstruct each sentence and tag words based on the sentiment it contains.

Goodboy's technical partner Mat Groves says: "We ran the sentiment information through an algorithm that would lay out the poster and render it using some custom-made webGL shaders. The cool thing is this all happened in real time!"

Users can also share memories through speech input in browsers that support the Web Speech API – another great enhancement in a site full of beautifully considered details. The site is no longer live, but you can view it on Razorfish's portfolio site at *40together.razorfishawards.com*.

> "Mindblown by this site, the technology is simply amazing! Explore 40 years of McDonald's memories. Kudos Razorfish"
>
> ADAM SAFAR (@XADUSX)

★ HTML, CSS, HISTORY API

IONICSECURITY.COM

nclud *nclud.com*

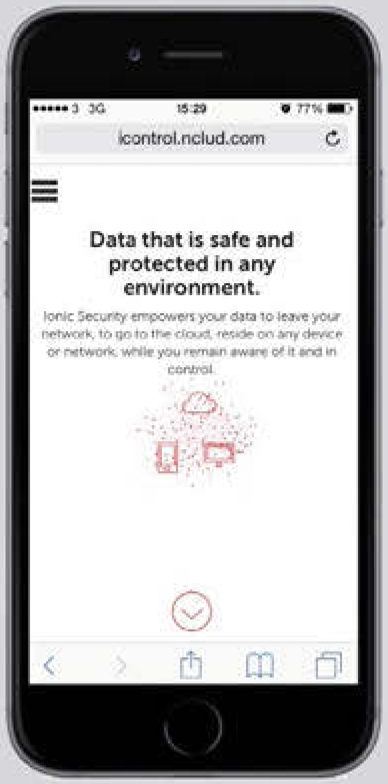
I'm not usually a fan of parallax websites that hijack the scroll, but I'll make an exception for this. Each swipe of the mouse moves between pages, presenting the content in a way that allows you to read it at your leisure, rather than frantically watching it rush past, as would happen on so many of the site's badly designed contemporaries.

Continuity between pages is provided by the little cloud of pixels that form a data visualisation. It's a clever effect, made with the Paper.js vector library.

"We only needed one HTML5 canvas element to achieve the effect," lead developer Ramsay Lanier says. "We used Paper.js, a super powerful animation library. The JavaScript to run the animation, including triggering changes on scroll, took only 400 lines of code."

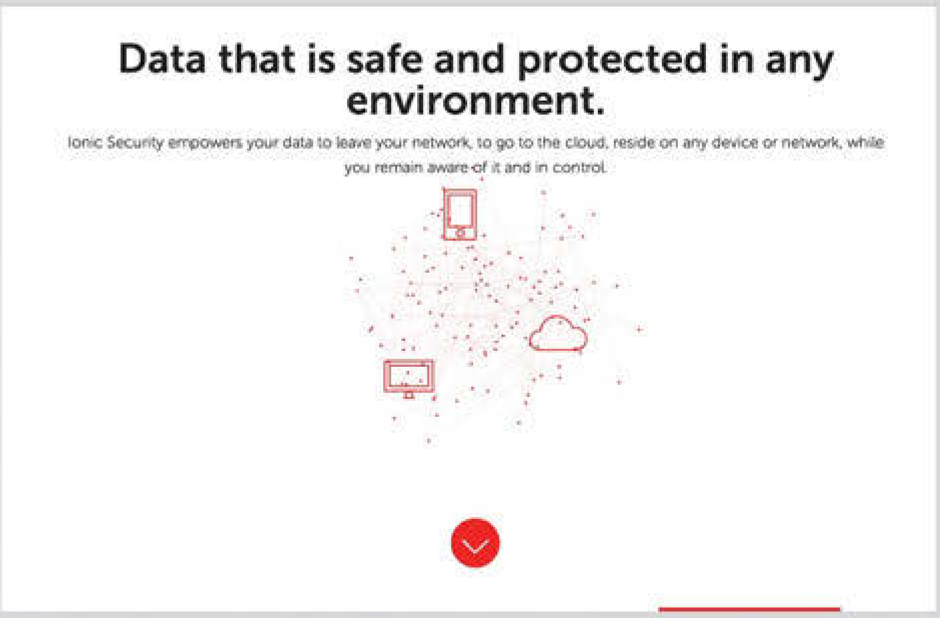
On tablet, where performance is more limited, the pixels are contained to a smaller area of the screen. On mobile they are gone altogether, ensuring the site is smooth and responsive to input.

The site is laid out with Isotope and features CSS3 animations and transitions. The overall effect is to ensure the emphasis is on the information, rather than just using flashy visuals for their own sake.



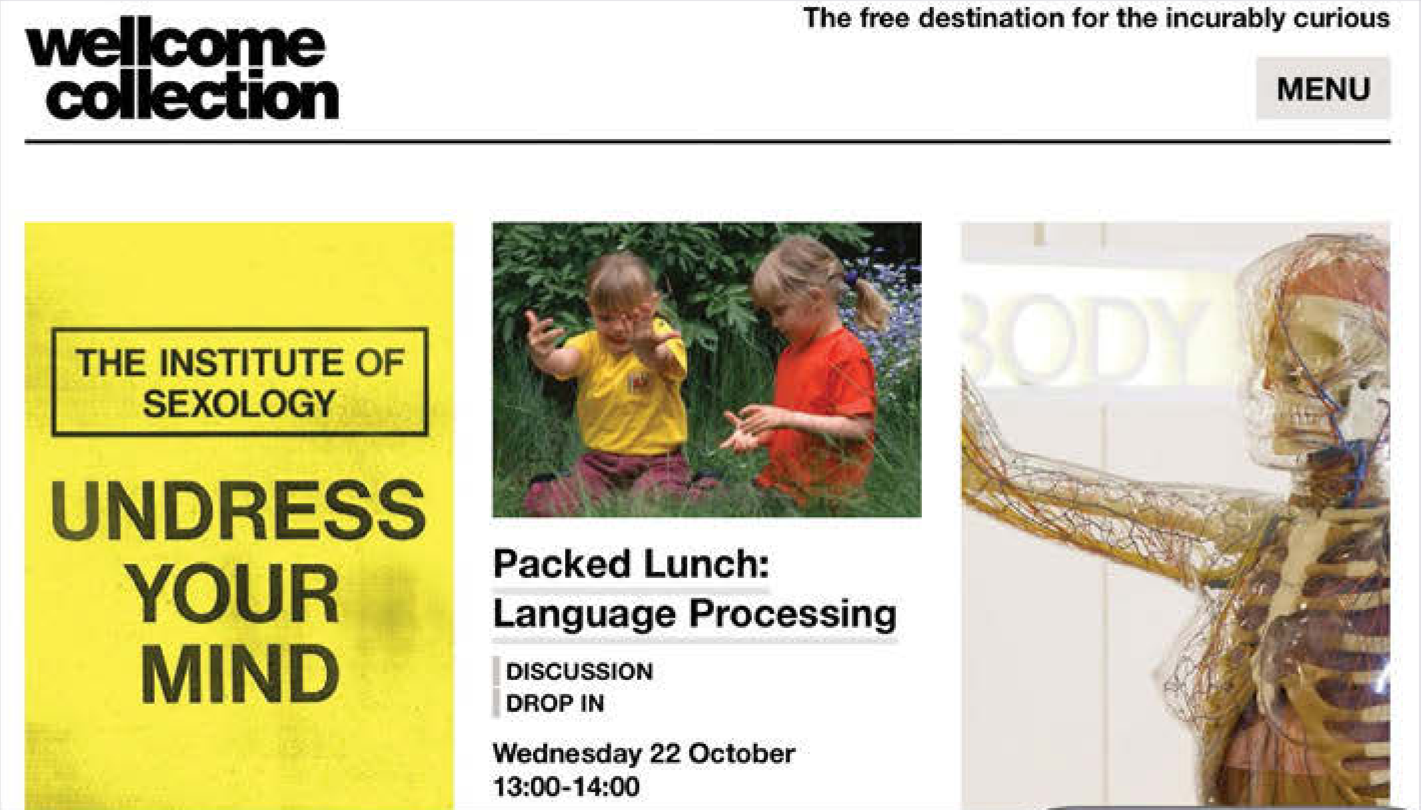
Data that is safe and protected in any environment.

Ionic Security empowers your data to leave your network, to go to the cloud, reside on any device or network, while you remain aware of it and in control.



Data that is safe and protected in any environment.

Ionic Security empowers your data to leave your network, to go to the cloud, reside on any device or network, while you remain aware of it and in control.



wellcome collection

The free destination for the incurably curious

MENU

THE INSTITUTE OF SEXOLOGY

UNDRESS YOUR MIND

Packed Lunch: Language Processing

DISCUSSION
DROP IN

Wednesday 22 October
13:00-14:00

*** DRUPAL, JQUERY, AMPLIFYJS, MASONRY.JS**

WELLCOMECOLLECTION.ORG

Wellcome Collection *wellcomecollection.org*

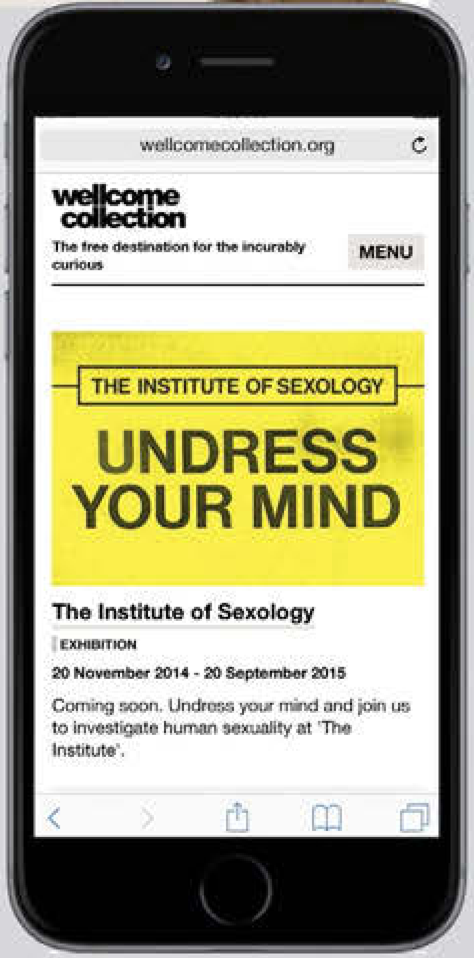
London's Wellcome Collection, a museum of medical artefacts and artworks, calls itself 'the free destination for the incurably curious', and its new, responsive website has the same attitude. Developed by the Collection's in-house team, the site promotes installations, events and publications, along with some exclusive digital content – including games.

The site is built on Drupal, which lead to some challenges. Developer Gareth Estchild says, "Drupal is a bit notorious for getting it to spit out what you want. The backend developers replicated the code from our HTML templates, which means we've been able to achieve good semantic markup."

The layout relies heavily on JavaScript: jQuery, jQuery UI and the AmplifyJS components library are in place to generate the common page elements, while Masonry is used to lay out the flexible grid.

Performance and findability were obviously foremost considerations in building the site: it's very quick to load even on low-bandwidth mobile data connections, and key information such as the address and gallery visiting times are present on every page.

UX designer Eleanor Ratliff says, "Simple little things, like flagging clearly on the listing pages when an event is fully booked, address some big user frustrations."



wellcomecollection.org

wellcome collection

The free destination for the incurably curious

MENU

THE INSTITUTE OF SEXOLOGY

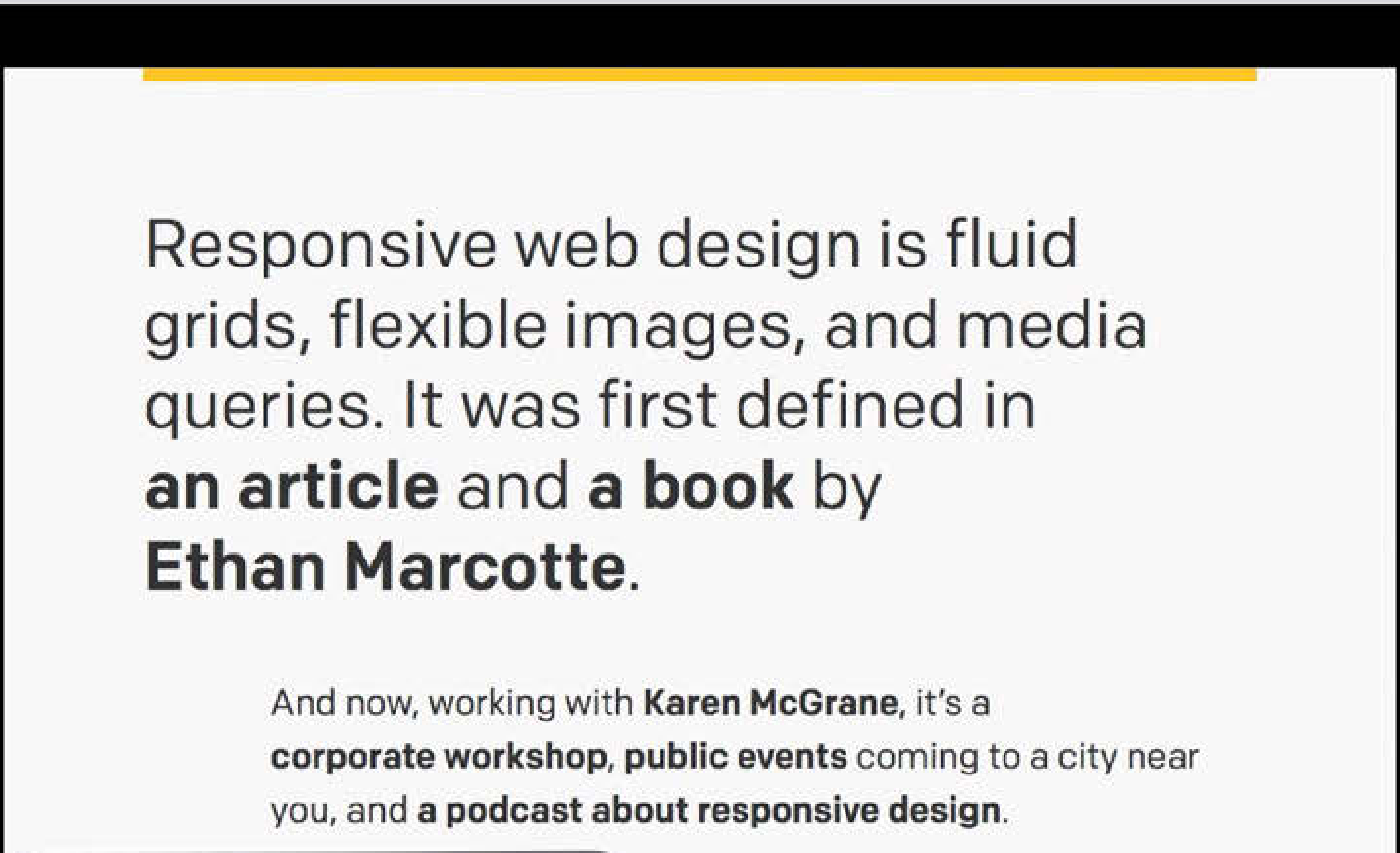
UNDRESS YOUR MIND

The Institute of Sexology

EXHIBITION

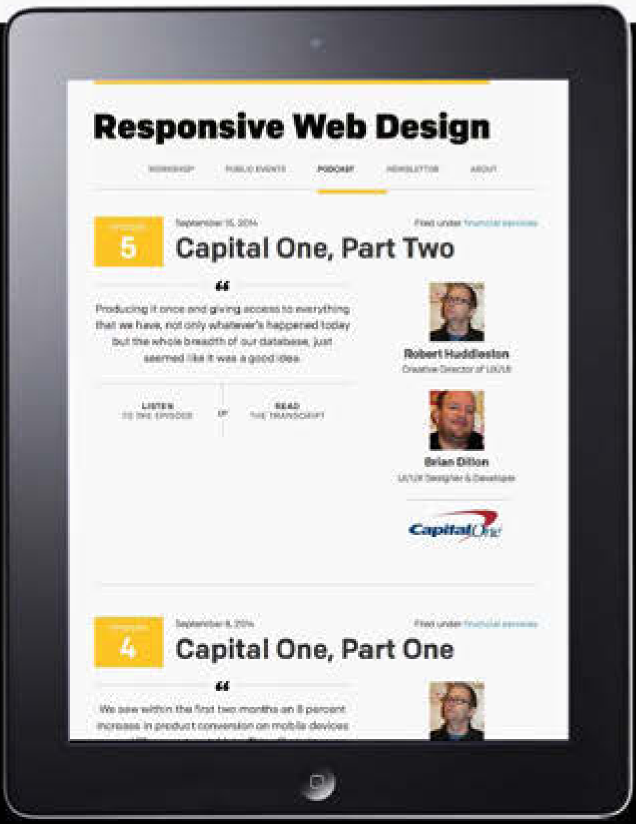
20 November 2014 - 20 September 2015

Coming soon. Undress your mind and join us to investigate human sexuality at 'The Institute'.



Responsive web design is fluid grids, flexible images, and media queries. It was first defined in **an article** and **a book** by **Ethan Marcotte**.

And now, working with **Karen McGrane**, it's a **corporate workshop**, **public events** coming to a city near you, and **a podcast about responsive design**.



Responsive Web Design

5 September 15, 2014 Filed under financial services

Capital One, Part Two

"Producing it once and giving access to everything that we have, not only whatever's happened today but the whole breadth of our database, just seemed like it was a good idea.

LISTEN TO THE EPISODE or READ THE TRANSCRIPT

Robert Huddleston
Creative Director of UX/UI

Brian Dillon
UI/UX Designer & Developer

4 September 8, 2014 Filed under financial services

Capital One, Part One

"We saw within the first two months an 8 percent increase in product conversion on mobile devices

✱HTML5, CSS3, RWD, GRUNTICON, JS ANIMATIONS

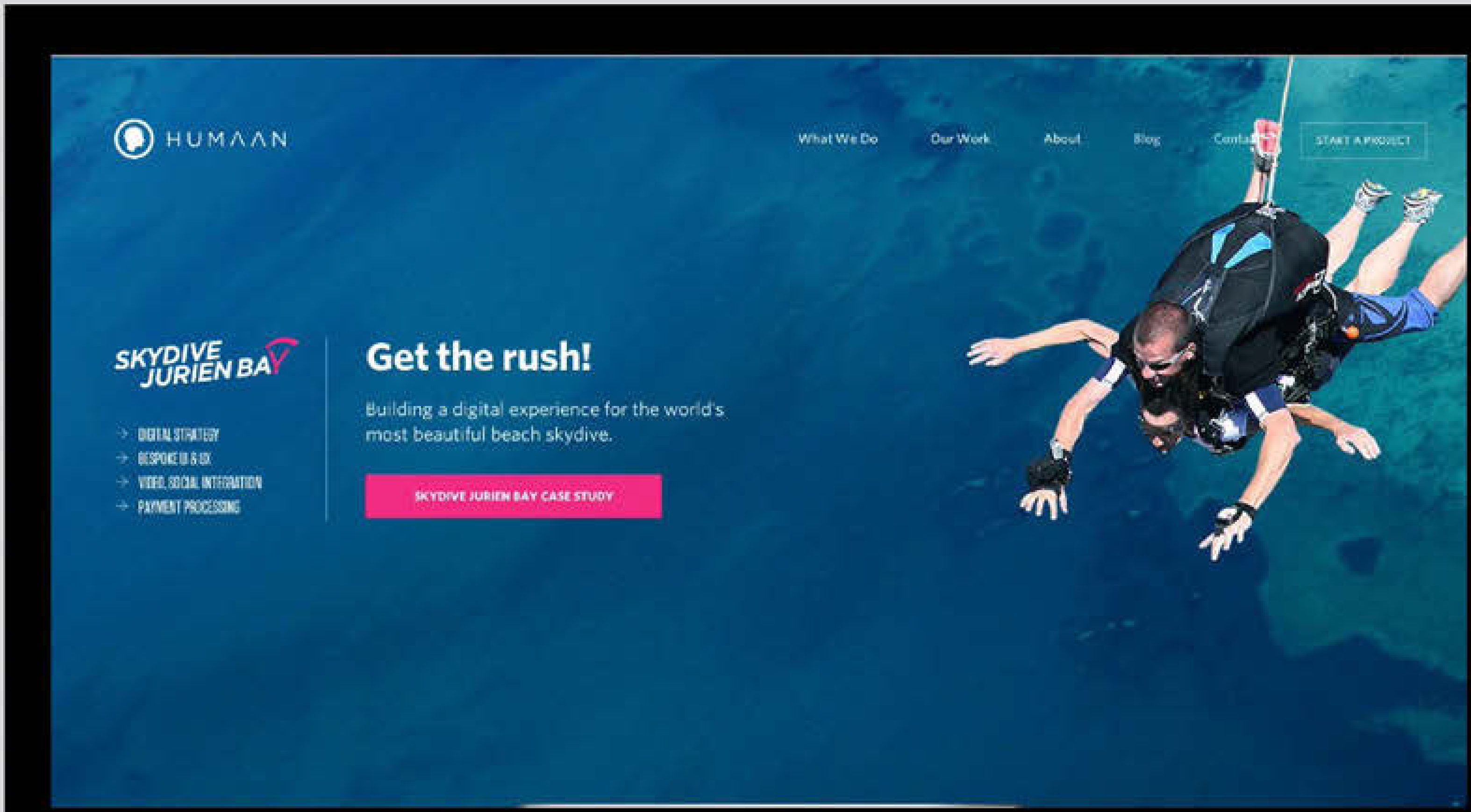
RESPONSIVEWEBDESIGN.COM

Ethan Marcotte *ethanmarcotte.com*
Karen McGrane *karenmcgrane.com*

Ethan Marcotte coined the term 'responsive web design' four years ago in an A List Apart article that was shortly followed by a descriptive book on responsive theory and practice. Now, together with Karen McGrane (author of *Content Strategy for Mobile*, published by A Book Apart), he's launched a site featuring workshops, public events and a podcast. And it's all about responsiveness.

Perfectly aligned with Marcotte's book branding, the site sets speed as its first concern. "Performance was a priority for us. I set up a simple performance budget, and it was actually freeing to view the site's design through that lens. There's still plenty of work we can do, but it's satisfying to see how quickly our site loads over a spotty 3G connection," says Marcotte.

"We're using Grunt to manage common tasks and a number of Filament Group-produced utilities to keep progressive enhancement and performance front-and-centre," adds Marcotte. He also mentions such tools as gruntIcon, grunt-criticalcss, CSS and JavaScript loaders. The result is a faultless example of responsive web design in action.



HUMAAN

What We Do · Our Work · About · Blog · Contact · START A PROJECT

SKYDIVE JURIEN BAY

- DIGITAL STRATEGY
- BESPOKE UI & UX
- VIDEO, SOCIAL INTEGRATION
- PAYMENT PROCESSING

Get the rush!

Building a digital experience for the world's most beautiful beach skydive.

SKYDIVE JURIEN BAY CASE STUDY

ANIMATION, CSS, HTML CANVAS, JS, GIFS

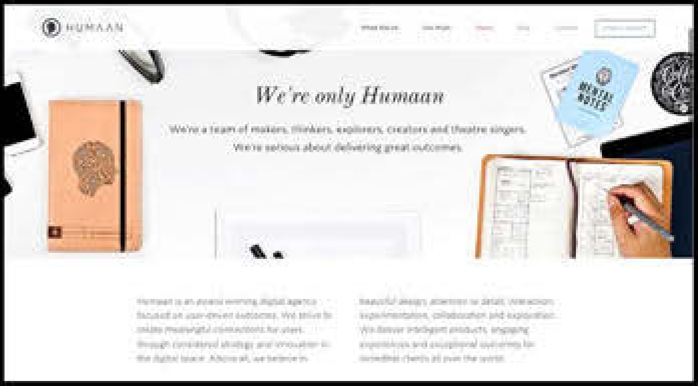
HUMAAN.COM

Humaan *humaan.com*

Humaan is an award-winning digital agency based in Perth, Australia. For its recent redesign, it stripped everything back to the essentials, with a focus on its story and impressive body of work.

Its new site reflects these values. There are some beautiful uses of animation throughout – the Humaan logo switches from 'Humaan' to 'Home' with a cute icon to match. There are also some more powerful uses: for example, when you navigate from the homepage to a project, the whole site rotates in the case study, which creates a beautiful full-page experience. Further animation is used within the case study pages themselves, and the content loads in softly and is easily digested. These are just a few little touches that make the site a pleasure to browse.

"After working through the development of our new direction, we wanted a website that better reflected our evolution as an agency, our voice and our focus on human interaction," says Humaan designer Kylie Timpani. "We incorporated a number of different technologies to bring the site to life, including CSS and SVG-based animation; HTML Canvas, JavaScript and good old animated GIFs. The ultimate outcome is one that we feel outlines our vision and communicates purpose, while also providing visitors with an exceptional experience."



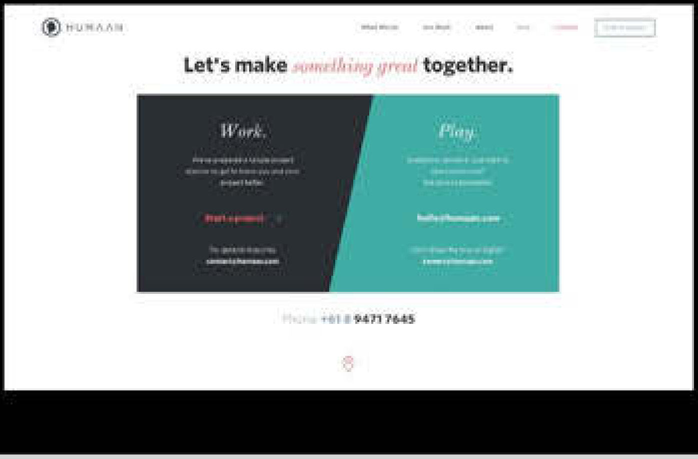
HUMAAN

We're only Humaan

We're a team of makers, thinkers, explorers, creators and theatre singers. We're serious about delivering great outcomes.

Humaan is an award winning digital agency focused on user-driven outcomes. We strive to create meaningful connections for users through considered strategy and innovation in the digital space. Above all, we believe in

beautiful design, attention to detail, interaction, experimentation, collaboration and exploration. We deliver intelligent products, engaging experiences and exceptional outcomes for incredible clients all over the world.



HUMAAN

Let's make *something great* together.

Work.

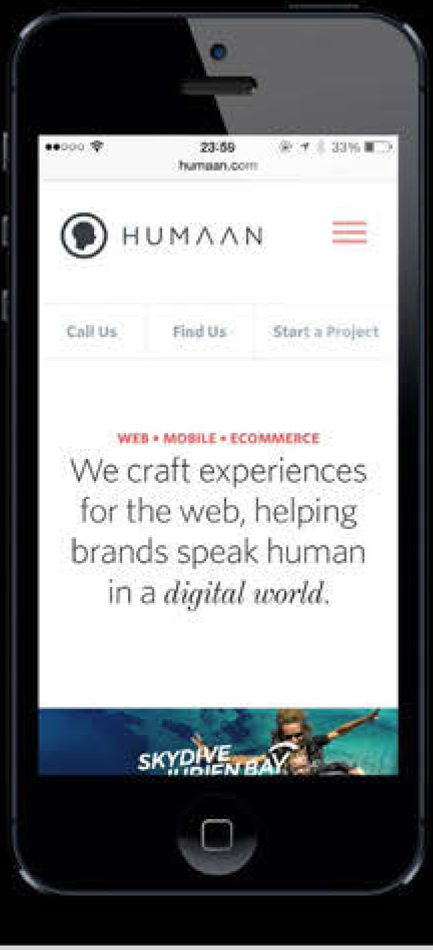
Start a project

Play.

hello@humaan.com

Phone +61 8 9471 7645

> “The Humaan site is beautiful; a great example of designing for real people and telling a story. Full of wonderful little touches, the site is packed with character and personal interactions”
>
> RACHEL SHILLCOCK (@MISSRACHILLI)



We craft experiences for the web, helping brands speak human in a digital world.

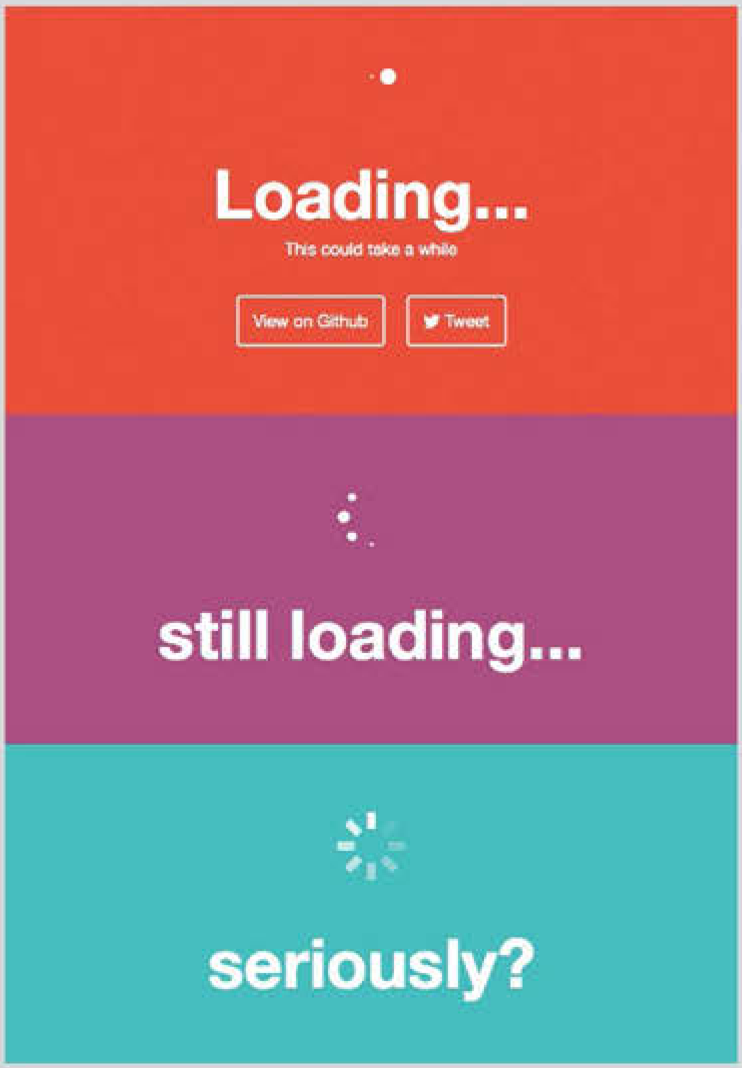
★SVGs, JAVASCRIPT

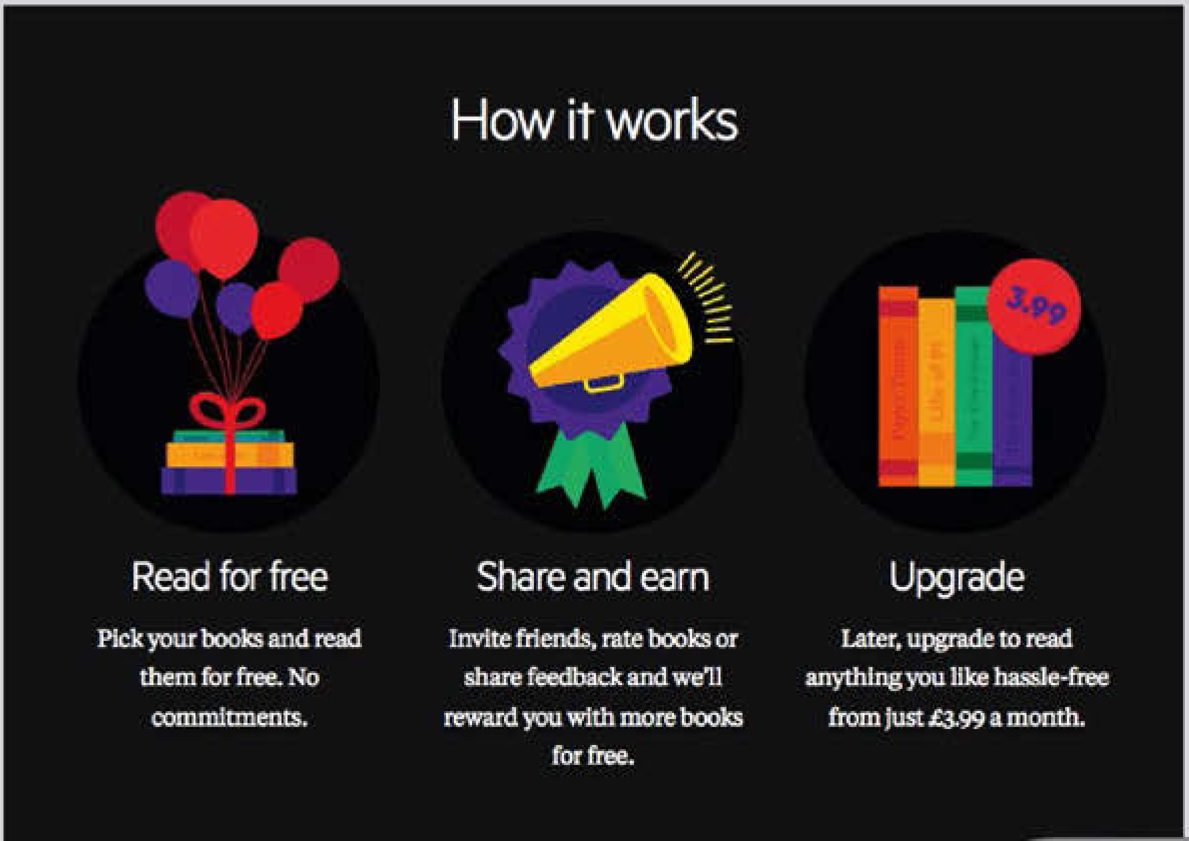
JXNBLK.COM

Brent Jackson *jxnblk.com*

SVG’s main domain so far seems to be an alternative for raster imagery and animation, but it can also be utilised in very interesting ways – including font replacement. Product designer Brent Jackson built Fitter Happier Text (*jxnblk.github.io/fitter-happier-text*) as an alternative to jQuery-based FitText (*fittextjs.com*) by Paravel. It still relies on JavaScript, but is significantly more lightweight and performant.

Jackson started playing around with SVGs before by creating a set of SVG-based loaders (*jxnblk.github.io/loading*). “This was mostly meant as a joke and an opportunity to learn about animated SVG,” he explains. Use with caution, though. “If you do intend to use these in production, be aware of limited browser support and be prepared to create fallbacks,” Jackson adds. Experiments like this prove that we’re still discovering the possibilities of what we can build with SVGs.





How it works

Read for free

Pick your books and read them for free. No commitments.

Share and earn

Invite friends, rate books or share feedback and we'll reward you with more books for free.

Upgrade

Later, upgrade to read anything you like hassle-free from just £3.99 a month.

*HTML5, CSS3, PARALLAX, GRUNT, RUBY ON RAILS, SASS, COMPASS, BOWER

BLLOON.COM

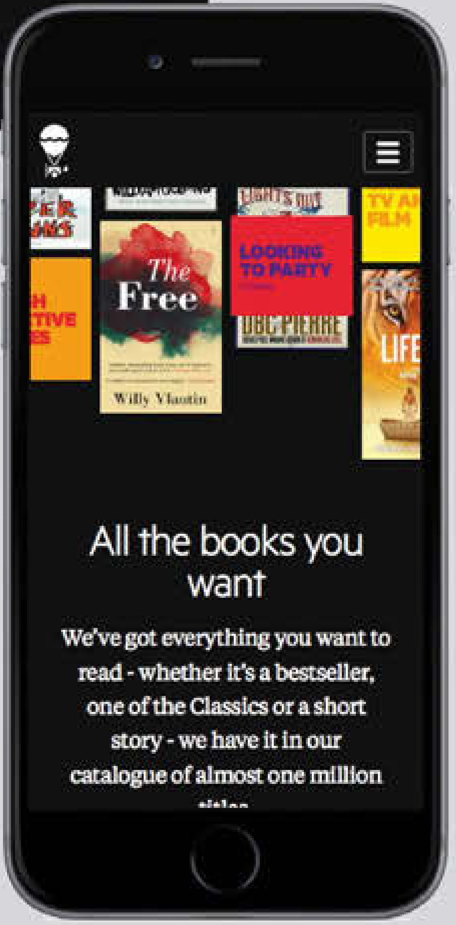
Edenspiekermann *edenspiekermann.com*

Blloon is a new ebook service that enables users to read for free, earning more 'pages' through sharing the app or rating their reads. Berlin-based Edenspiekermann was the agency tasked with developing the branding and digital product.

"With Blloon, we wanted to create something that differentiated from the ocean of startup sites out there with looping background videos," says creative lead Paul Woods.

Matt Berridge, lead developer on the project, explains that the team did some groundwork at the start of the project to streamline development. "We integrated Grunt to automate tasks such as image processing, optimisation and Modernizr compilation," he says. "We also integrated Bower into the asset pipeline as well as Autoprefixr, saving us time as the project went on." The site makes use of a two-tier Javascript set-up based around the BBC's 'Cut the mustard' test. The team created a 'core' JS experience, and progressively enhanced things from there.

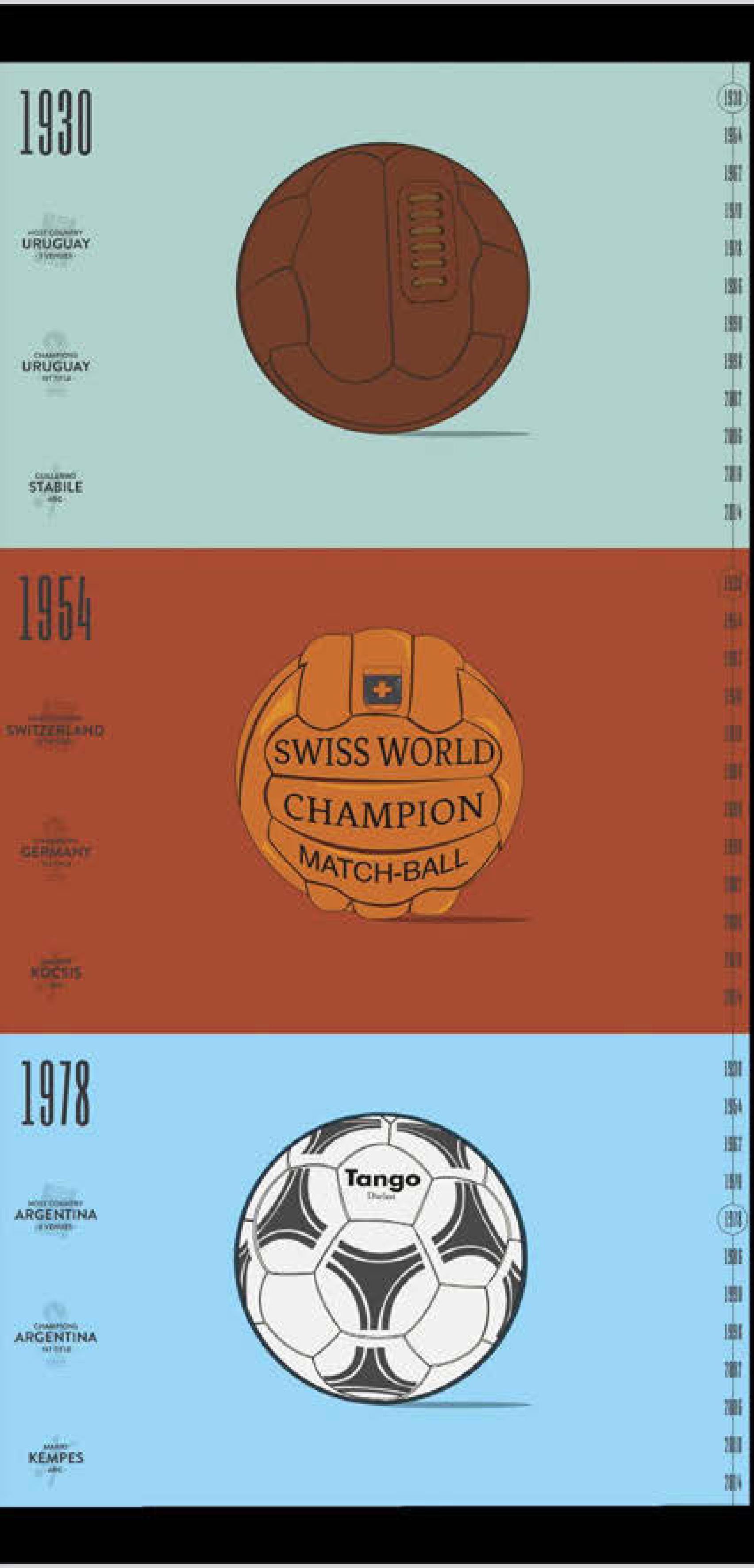
"We utilised open source scripts like Typer.js for the hero module, alongside a JSON feed containing titles, quotes and book covers (loaded via Ajax) from the Blloon book database," adds Berridge. "On the book detail page, adaptive-backgrounds.js allowed us to calculate a suitable background colour given the book's cover."



All the books you want

We've got everything you want to read - whether it's a bestseller, one of the Classics or a short story - we have it in our catalogue of almost one million titles

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL



★RESPONSIVE, ANIMATION, BOOTSTRAP, WEB FONTS

WORLDCUPMATCHBALLS.COM

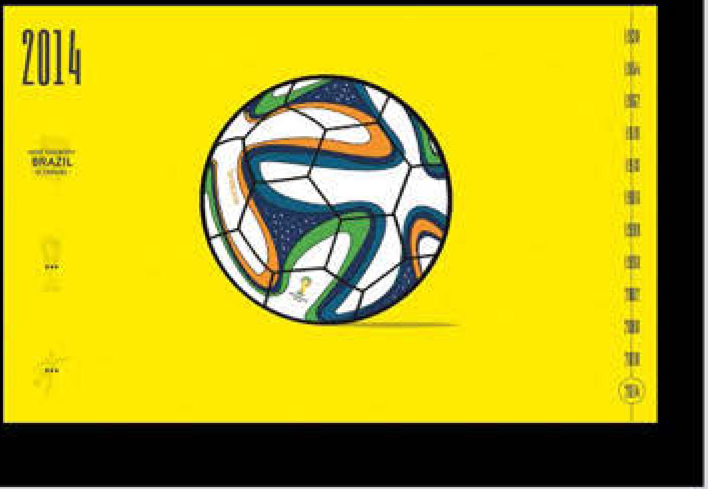
150UP *150up.com*

To celebrate the 2014 FIFA World Cup in Brazil, the football fans at Milano-based studio 150UP launched World Cup Matchballs; a fun, responsive showcase of football designs used in past tournaments. The website makes use of animations to bring the vintage illustrations to life.

The single-page site was designed in Sketch – a first for the agency, which had previously used always Photoshop. "We found Sketch very useful, quick and fluid. It also gave our designers and programmers the opportunity to work together, speeding up the process," says founder Davide Colla. "Our aim was to give the user the best experience, both from a design and programming point of view."

Bootstrap was used for layout, while JavaScript was written from the ground up for the scrolling timeline. CSS3 animation was added to make the balls react to the scroll. The site utilises both Typekit and Google Fonts to suit the vintage-style design.

"We thought the World Cup would be a big bonding topic, and we wanted to give our best support to our Italian national team as well," Colla recalls. "Unfortunately, that didn't go well for the Italian team, but we tried our best!"



THE FUTURE OF JAVASCRIPT

★FUTUROLOGY

BUILDING THE SENSORY WEB

With the rise of a new generation of web-connected devices, designing for the screen alone will become a legacy activity, says **John Allsopp**

A lifetime ago I wrote 'A Dao of Web Design' (*alistapart.com/article/dao*). At the time, it made a bit of a splash, then lay largely forgotten until Ethan Marcotte penned 'Responsive Web Design' (*netm.ag/rwd-244*) which begins by quoting this from my article:

"The control which designers know in the print medium, and often desire in the web medium, is simply a function of the limitation of the printed page. We should embrace the fact that the web doesn't have the same constraints, and design for this flexibility. But first, we must 'accept the ebb and flow of things.'"

Now 15 years later, in some ways our concerns as web designers and developers are much the same as they were in early 2000. They focus on the surface of what we design and build – on the screen, on what appears on it, and on how the user consciously interacts with it.

Now, that surface is undoubtedly important. But I increasingly see it as only a small portion of what we can do with the web; what the web enables.

Photography: Drew McLellan

To continue with the metaphor, beneath the screen lies a device, with all kinds of capabilities that we developers are increasingly being given access to. Gyroscopes and accelerometers and geolocation capabilities allow us to get a sense of where in space a user is, and their physical context: are they sitting, standing, running; travelling by plane or car or bicycle? Are they in distress or relaxed?

Our mobile phones and tablets have cameras and microphones, and will increasingly contain an array of sensors that can detect temperature and air pressure and quality. More specialised devices (connected fire alarms, for instance) can already detect the carbon dioxide, carbon monoxide and other gases in their environment.

Beyond the device lies a network of services it can talk to, send information to, and receive information from. In real time. From almost anywhere, almost any time.

THE SHAPE OF THE FUTURE

This, to me, is what the web of the future looks like. Not merely screens being driven by humans looking for information or entertainment, but a vast array of devices con-

The web of the future is not merely screens driven by humans looking for information, but a vast array of devices constantly sensing the world

stantly sensing the world: how fast is this car travelling right now? What's the temperature of my house? How many calories have I consumed today?

The challenge for us as developers and designers for the web becomes less about screens and pixels and buttons and much more about how the web augments our lives, both actively and passively; how it makes us know ourselves and our homes and workplaces and environments better.

Now, if that all sounds a bit woolly, it is. Partly deliberately, and partly of necessity. If the future is to be interesting, we can't really know it, or predict it, except, as in Alan Kay's famous observation, by inventing it.

But if I were advising young players at home about what they should be learning about to give them longevity on the web, I'd say: learn JavaScript. Not only is it the language of the browser but increasingly, with Node.js, the language of the server, and is even found directly on devices.

And I'd start exploring the at times dizzying array of low-cost network-enabled devices and sensors, like NinjaBlocks (*ninjablocks.com*), Tessel (*tessel.io*) – which you can program in Javascript directly on the hardware – and dozens of others.

And I'd explore Node.js (forked to io.js and node), as the glue to connect this vast array of devices to one another, and to services that can gather and make sense of the vast amounts of data they will generate, and then enlighten our users about their habits, practices, health and lives.

Which all seems rather daunting – new languages and devices and services, not to mention skills and knowledge – but as always with the web, each of these is in a way a small piece, and those pieces will be loosely joined. There's no need to stop what you are doing today, because the screen will always be an important part of any system that humans engage with, but designing and developing for screen-based systems alone will, just as designing and developing for desktop browsers alone has become a legacy activity.

Don't be daunted: after all, you've come this far on the web, and over time acquired a broad range of knowledge and skills, whatever you do day to day. Rather, be excited that we stand at the threshold of something amazing. Quite what it is we can't be sure – but to me, that's the best part of all.

★ PROFILE

John Allsopp's 'A Dao of Web Design' has been described as "a manifesto for anyone working on the web" by Clearleft founder Jeremy Keith. Follow John online: @johnallsopp

JOHN ALLSOPP

Words by Martin Cooper **Photography by** Xavier Ho

Forget jetpacks and robots – John Allsopp has gazed into the future and seen a world of JavaScript, movement monitors and tools for OAPs. And for the 'Dao of Web Design' author, that's an outcome well worth fighting for ...

INFO

job: Developer, author, conference organiser

w: *johnfallsopp.com*

t: @johnallsopp

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Future-powered
"What interests me is taking what people are doing naturally and unconsciously and turning those things into interactions ... When we think about the future we have these huge *Minority Report* fantasies. It's much more incremental"

The Internet of Things. The connected web. The sensory web. Call it what you will, the web is changing and, quite soon, it will be all around us. Amazing devices are coming. Screens here, there and everywhere. The web will envelope us, and for web workers the possibilities are limitless. But that's not necessarily the point ... or so says John Allsopp.

"People think the future is about jetpacks," Allsopp asserts, waving his hands. "You know, sometimes we believe the future has to be awesome, and if it's not we're disappointed." The future, according to Allsopp, is likely to be rather more *Last Of The Summer Wine* than *Minority Report*.

Think about it, he challenges: "The fact is, if you can keep a significant percentage of old people in their home for another year, the savings will be huge." To achieve this doesn't require complex, impressive systems – all we need are basic, web-enabled monitors that record and broadcast heart rates and the like. "It's not just about saving money. They'll have dignity, they'll have control. And that could be done using a humble mobile phone. What greater opportunity? What greater challenge?" That, to Allsopp, is a future worth pursuing, and there's not a jetpack, exoskeleton or internet-enabled fridge in sight.

If anybody should be able to predict the future, Allsopp is more qualified than most. He is, after all, the man who just might have coined the inglorious term 'Web 2.0' (he can document his usage of it before Tim O'Reilly popularised the phrase). "It was back in early 2005 ... I wanted to give a sexy name to the web stack – to HTML, CSS and the DOM. I sent an email to a whole bunch of people involved with the Web Standards Project saying 'we need a concept, we need a brand'. But you know, the irony is, I used to blog everything but I didn't blog this!"

GETTING STARTED

Allsopp is also feted as one of responsive design's founding fathers. His essay, 'A Dao of Web Design' (*alistapart.com/article/dao*) was published in 2000. In it he encourages designers to let go of print's rigidity and embrace the web's fluidity. Like in martial arts, strength, he tells us, can be found in relaxing. Ethan Marcotte cites Allsopp's essay as one of his key inspirations.

Office culture From design classics to dongles, the Web Directions office – which hosts meetups, hack days and other industry events – has collected all manner of paraphernalia

These days, Allsopp teaches, speaks, makes, tinkers, writes and runs the Web Directions conference (*webdirections.org*). Based in Australia, Directions this year celebrated its 10th anniversary.

However, Allsopp's technical career dates back to the days before the web was even around. He was working as a software engineer in the late 80s when he built a Mac-based platform for creating knowledge-based systems. "We decided to sell the system online, which was very unusual for the time. To do that I had to understand the web and build websites. That's how it all got started," he finishes. Looking back, he says, if he'd understood how big the web was going to be, he would have applied himself to the project more fully.

> "Preprocessors exist to make developers' lives easier. There's no semantic benefit. Everything just ends up as CSS"

FAST-FORWARD

So, what's Allsopp's take on where the industry stands now? Or, to put the question another way, how far have we as a community come from the days when 'A Dao of Web Design' shook the web?

"You know, I think we have a good understanding of the challenges – from a visual design perspective – of designing for a screen," he replies. "There's a degree of maturity and we're increasingly getting the solutions right. I think the work on the picture element by the Responsive Images Community Group was fantastic. For the first time, the people who used the technology – people who work at the coal-face, making things, web designers and web developers – they got up and they shaped what a technology should actually look like, and how it should work. And it was done in the face of objection."

He recalls a lecture he gave recently, where he discussed CSS3 factors – like border-radius, gradients and drop shadows – which are now almost obsolete. "We've been using them for 20 years, though we might have been doing them in a horrible way. The problem is, the standards bodies move so slowly, by the time something arrives in the browsers it's passé," he elaborates. "What I like about the community-driven stuff – like the picture element – is that features come along more quickly."

Concluding his point, he says: "Web design is just web design. It's not responsive design. If you're not a good responsive designer, you're not a web designer. Responsive is just table stakes now."

KEEP IT SIMPLE

Allsopp is, however, keen to draw a distinction between the visual parts of products (he calls this the 'surface layer') and site's mechanical underpinnings. This leads to certain frustrations with today's web tools.

"I'm a curmudgeon," Allsopp laughs. "I have a background in software engineering … and in software engineering, there's a difference between simplicity and easiness." Developers often, Allsopp

explains, want to make their lives easy by reducing keystrokes and banishing basic, repetitive tasks. "So, we build these really complex stacks of technology to make our lives easier. The problem is, we often build things that are much more complex because that's the price we pay for an easy life."

To illustrate his point, Allsopp points to CSS preprocessors. "They exist solely to make developers' lives easier. They are syntactic sugar. There's no semantic benefit. They can't do anything, as everything eventually just ends up as CSS. Sure there are many cases where preprocessors are valuable, but there are many cases where they're not."

One of the challenges preprocessors bring to the table, he says, is added complexity. "You've added complexity to the syntax and [the developer] also needs to know both CSS and Sass. Then, you need build tools to process your Sass into CSS, and then you put it into your browser and you want to debug it. Now you need source maps … We're creating an awful lot of complexity for the ease of the developer."

And why is that a problem? "It's a problem because the more complex a product is, the more costly it is to maintain. The thing they learned in software engineering a long ►

Left Allsopp doing what he loves best – coding **Above** Web Directions' feature blackboard wall has appeared on websites, book covers and serves as a great 'old school' tool for brainstorming

time ago is that 80 per cent of the cost of a system is maintaining it. If you build a complex system, you're adding to the cost."

Along with adding cost, complexity also brings other problems to the table – not least concerns about the availability and longevity of the other systems your product relies on. "If your system is going to last any non-trivial length of time and you've got dependencies on version 'x' of Sass and version 'y' of Ruby on Rails, you wind up with a whole lot of technical depth that you've got to pay down at some point. You also need people who know all of the technologies."

He concludes with a warning against always grasping for the latest and most exciting tool. "I guess that's a way of saying we rush too quickly to new shiny things, rather than looking at the core, simple things like HTML, CSS and JavaScript. We're making things too complex and I think we should be careful."

ENTER THE FUTURE

Our conversation moves back to future. "Today, almost all interactions are conscious. Whether its touch, or mouse or voice, it's still conscious. We're building systems that are dumb and passive. They're sitting there in a loop. That's how GUI programming works. There's an event loop and it waits until it gets an event and it responds. So, when we think about interaction, we think about a computer sitting there, waiting for us to touch it."

The thing is, we humans can't deal with too many thoughts at once. So if we want a computer to do something, we have to stop what we're doing and thinking about. For the most part, Allsopp argues, interaction is a block.

What interests him most is the idea of unconscious interaction. People are walking around with a phone in their pocket, and all the time they're creating a trail of data, he explains. For example, the pattern of walking could suggest, when analysed, that the person has the flu. "What interests me is taking what people are doing naturally and unconsciously and turning those things into interactions," he finishes.

> "Learn JavaScript. I see a huge demand for it. It's a common language that we're seeing everywhere, even in hardware"

The future, though, never looks quite like the future. "When we think about the future we have these huge *Minority Report* fantasies. It's much more incremental. I think we get overly ambitious, we take today and put jetpacks on it."

FOUNDATIONS

And what's going to power this future? Allsopp's prediction is, once more, not quite the stuff of movies. "JavaScript. Learn JavaScript. I see a huge demand for it, it's going to be the lingua franca. People criticise it for all its problems and limitations. It's a common language that we're seeing everywhere, even in hardware now. You script Mac OS with JavaScript now. And, you know, lots of web designers have picked up some HTML and CSS along the way, and then just stop at JavaScript. People think: 'Programming? How can that be interesting? It's scary. It's hard.' Don't think like that. It's logic and humans are logical. Our brains are hardwired to do 'if', then 'else' stuff. It's not much more complicated than that."

REWIND

Recently, **net** magazine celebrated its 20th anniversary. And if there's one thing that sums up the last 20 years, and will hopefully set us up for the next 20, it's this observation from Allsopp: "It's the 'World Wide Web'. It's got that 'World Wide' bit in its name, and that's important. It underlines the web's values, which are: openness, interoperability and access."

For Allsopp, working on the web is like forming a pact. "The web has been very good to us – it's given us a good life, it's given us interesting problems to solve, it's given us a great community of people to work with, right around the world … We're immensely privileged. The very least we can do to pay the web back is to understand its values and to honour them."

USER EXPERIENCE

THE INTERNET OF THINGS

Dan Frost considers how the Internet of Things will impact user experience

Heard of the Internet of Things? If not, there's a good chance you own something that connects to your phone, Wi-Fi, car or some other digital part of your life. The Internet of Things is a world where your unconnected stuff (cars, doors, buildings, shoes) becomes connected. This is all becoming possible because the price of computing and connectivity has fallen to the point where stuff can be hooked up for a tiny additional cost. Increasingly, manufacturers can connect their products for as little as a dollar a year.

At this falling price, anything can be connected, from shoes and tables to lamps and lights. Drugs are being tagged with RFID. Food can be tracked at the molecular level. We are going to see more than fridges connecting to the internet. But there is a problem, and this is where you, user experience designers and developers come in.

The gap between technology and users is closed by really good experience design. This goes under different names: industrial design for physical products, user experience for digital products. Even print layout can be viewed as experience design. For example, this page is laid out and typeset to make the information easy to read and ensure it forms part of **net**.

DEFINING USE CASES

The problem is that we don't know what the use cases will be. We have a sweet shop of technology and no clear ideas what will be a great product. To compound this, we're still at the very early adopter stage where the people buying the exciting new stuff will tolerate a complex user experience because they love new stuff. To take this technology to the masses and build a next generation of connected products, the experience design has to feel beautiful. Great products just fit. Using them feels obvious. The design looks inevitable.

We don't know what the use cases will be. We have a sweet shop of technology and no clear idea what will be a great product

GOOD PRODUCT DESIGN

No one knows which use cases will take off because we don't really have the examples and content to work with, but we can learn from good product design. As this new world is the merging of digital and physical products, we need to learn from both domains. What makes a beautiful and useful physical product? What makes an engaging digital product?

You can, and should, buy and hack together simple IoT projects using a Raspberry Pi or Arduino. Once you have your connected robot or sensor, push yourself to design an experience that feels like a finished product. See if you can put it in the hands of a non-technical user in the same way you would user-test a website. See how many people you can convert to users of your physical-digital product.

When we started designing our first product we had many conventions to challenge, especially coming from the hardware design. We kept having to push ourselves to make the experience simpler and more obvious to earn trust.

What emerged were a few rules for designing connected experiences across digital and physical products. These became our Manifesto for the Experience of Things (*seamlessli.github.io/mfteot*) and help frame experience design decisions, including: design for direct, transient and shared ownership; don't carry over the experience unnecessarily; design for scale of devices, not just of users; consider the user's life with tens or hundreds of devices; and imbue ownership so users trust and use your product. You earn the right to the user's data by giving them ownership of the product.

You can read and contribute to this on GitHub (*github.com/seamlessli/mfteot*). We probably don't have the answers for the connected world in 10 years' time, but by learning from the best physical and digital products, we've got a start.

What do you do now? Get yourself a Raspberry Pi project, push to make the experience as polished as possible and fork the manifesto with everything you learnt about the new world of connected product design.

PROFILE
Dan Frost is technical director at digital agency 3ev (*3ev.com*) and cofounder of IoT startup Seamless (*seamless.li*)

*WEB STANDARDS

THE EXTENSIBLE WEB MOVEMENT

Steve Klabnik introduces the movement set to invert the web standards process and speed up development times

The winds of change are blowing in the web standards space. While the standardisation process does work, it still has problems. In addition, the nature of the web is changing – and with it standards bodies are figuring out the best way to make sure it grows in such a way that allows for continued growth in the future.

This movement is called the Extensible Web, and it has a manifesto (*extensiblewebmanifesto.org*) that has been signed by over 900 developers from around the net – including high-profile names like Google's Paul Irish and JavaScript creator Brendan Eich.

Extensibility has been one of the main properties that software architects have pursued for decades, and it's also one of the hardest. The manifesto outlines its strategy to accomplish this task:

Today, most new features require months or years of standardisation, followed by careful implementation by browser vendors, only then followed by developer feedback and iteration. We prefer to enable feature development and iteration in JavaScript, followed by implementation in browsers and standardisation.

You see, the current standards process is driven by browser vendors, because they are the only ones with the ability to significantly alter the browser. This leads to a standardisation process where the majority of developers are only introduced after all of the hard work has been done and the feature has been released.

If anyone who knows JavaScript could extend their browser with it, this process could be inverted: your everyday JavaScript developer could create new browser features, which could then be standardised and implemented natively by browser vendors. New features often require a number of iterations to get right – the easier we make iterations of the web platform, the better features we'll get.

BIGGER IS BETTER?

The Extensible Web's strategy is very similar to Guy Steele's. In 1998, Steele gave a keynote address at the ACM's OOPSLA conference. His talk, 'Growing a Language', asks the question: If I want to help other people write all sorts of programs, should I design a small programming language or a large one?

Your everyday JavaScript developer could create new browser features, which could be standardised and implemented natively by browser vendors

A small language can be implemented and learned quickly, but it can be difficult to get real work done because you don't have a lot of tools to work with. A large language, on the other hand, may take a while to learn, but gives you great power in solving whatever problems the language's designers thought were important. However, if a totally new problem arises, a large language can be hard to modify, since there are so many other features to consider.

So a small language can't do the job and a large language takes too long to get off the ground. Are we doomed to use small languages with many warts because that is the only kind of design that can make it?

To address this, Steele draws a comparison between Lisp and APL. He asserts that APL was hard to extend, since primitives used all kinds of Unicode characters, but Lisp was very easy to extend, because primitives looked indistinguishable from user-defined code. This property meant that Lisp ended up being used much more often, since it was able to include extensions that made working in the language much easier.

I like to think about this property as a comparison between Common Lisp and PHP. Both languages started out with no support for Object Oriented Programming, but include it now. Common Lisp, being a Lisp, is incredibly easy to extend, while PHP is not quite as easy. CLOS (Common Lisp Object System) was originally a library, and not part of the language itself. It could be worked on, tried out and improved for years before making it into the standard for the language. PHP, however, had to have its object orientation support, introduced in PHP 3, fully implemented in the reference interpreter before anyone could try it out. This led to lots of changes, all the way up to PHP 5.

Today's web browsers operate like a large, non-extendible language. They provide lots of tools for developers, but solving new problems can be painful. If the Extensible Web comes to pass, browsers will still be large, but will be extendible. This extensibility will bring about the inversion of the standards process, but also allow more features to be developed more quickly, which will greatly benefit all of the web's users. The web recently had its silver anniversary, and I'm excited to see it continue to evolve.

PROFILE

Steve (@steveklabnik) is a prolific open source contributor, Rails committer, Rust documentor and hypermedia enthusiast. He's also written several books, including *Rust for Rubyists*

DEVELOPMENT

SIMPLICITY MEANS SPEED

As frontend development advances, file sizes are rising alongside load times. It's time to spare a thought for the slower browsers, says **Scott van Looy**

There has never been a better time to be a frontend developer. HTML5 and CSS3 have allowed us the flexibility to design user interfaces consistently across a wide variety of platforms, whilst JavaScript has been elevated from a scrappy scripting language into a fully fledged programming language with its own ecosystem, development environments and developer community. We are now equipped to do battle with native apps – indeed, the line between what's native and what's not blurs ever more with each browser iteration.

In our quest to become fast-yet-full-featured, I feel we're missing some tricks. How often have you added a library to your frontend code? And then added a framework? Maybe a selection of plugins. A dash of responsive grid. Some shims and polyfills. And finally a sprinkling of images?

Suddenly, you mysteriously have hundreds of kilobytes of JavaScript and CSS to download. Libraries and frameworks built on other libraries, full of hacks for older browsers, relying on gzipping and minification to make things small. And that's before you've even reached any of the content you want your users to see.

You expect your pages to run quickly on those older browsers, or on mobile devices with little available memory. You complain when they don't, that it can't be done, that mobile browsers are too slow, that JavaScript in phone browsers isn't fast enough, that the DOM isn't fast enough. Sound familiar? Well, there's another way.

QUICK AND SIMPLE

A technique that seems to have been eclipsed somewhat is that of progressive enhancement. Use the server to create your pages, don't send everything to the client and make them assemble it. Write valid, semantic HTML. Target your slowest browsers and do everything you can to make them fast. If you have to support IE8, code with a copy of IE8 open. Test everything for speed and performance.

Once you have an actual page being rendered by the browser, you won't have to worry about JavaScript or the DOM being slow on mobile. Keep it simple. Keep it fast.

Use the server to create your pages. Write valid, semantic HTML. Target your slowest browsers and do everything you can to make them fast

Once you have a really fast template that works without JavaScript, you can then enhance the content with a sprinkling of JavaScript.

Here are my top tips for building the fastest possible websites:

- Make as few requests as possible. Ideally one HTML file, one CSS file and one JavaScript file. The icons and visual assets that constitute your user interface should be small, optimised and embedded as Base64-encoded images inside your CSS where possible.
- If you're using templates to construct HTML on the fly within the browser client, make those templates work on the server as well as on the client. Twitter spent a fair amount of time removing its old hashbang scheme for tweets, turning tweets into physical pages on the web. The company claims this dropped page-load times by 20 per cent.
- Load only what you need to see, when you need to see it. Sounds obvious, but often you'll have a long, scrolling page and all the images on that page will be set to be downloaded at once. This leads to a cascade effect – web browsers only open a certain amount of connections at a time to a specific host, so only a certain amount of your images will be downloaded at once. Use CSS to show background images when they're needed.
- With photographs and retina images, you can often get away with using 30 per cent quality in Photoshop's 'Save for web' feature, and doing the resizing in the browser. You end up with images that are almost the same file size as the normal ones, saved at 70-80 per cent quality, but with a higher resolution.
- If you use PNGs with alpha transparency, you can often get away with changing them from 24-bit to 256-bit colour palette using software such as ImageOptim with no discernible loss in quality but sometimes more than 50 per cent smaller file sizes.

With this approach, you get some added extras. SEO for free. Older clients will, for the most part, work too. Same with screen readers. It'll be trivial to add ARIA accessibility support. On top of that, you get the fluffy feeling of being a good net citizen. ◘

PROFILE ★

Scott (@svanlooy) has been creating websites since 1995. He has spent the past 7 years at AKQA London and Berlin, specialising in JavaScript CSS and HTML

★JQUERY

MOVING ON FROM JQUERY

It changed our world, but we don't have to rely on it forever. **Zack Bloom** argues life could be better if libraries stopped depending on jQuery

When jQuery was first introduced, it changed the world. It gave us all the ability to refer to elements on the page from JavaScript, like we could in CSS. Not only that, but it gave us an API for these elements that was miles ahead of what the browser had. We could hide them, add and remove classes, and even write our own plugins to do unprecedented things.

It also hid much of the complexity of dealing with that era of browsers from us. Through painstaking fixes, all of the various bugs that those browsers included were worked around, until you could be confident that if something was done with jQuery, it would work.

Before too long, jQuery became the vernacular of the internet. Scores of people learned it before they learned the DOM itself: it became an implied dependency in every library and tool out there. jQuery was ever-present.

MOVING ON

But things have moved on since then. jQuery thrived in a time when browsers were broken and slow to change, but that itself is changing. Since IE8, browsers have included the `document.querySelector` method, which gives us all jQuery's killer features right out of the box. IE9 gave us `addEventListener`, freeing us from `onclick` and `onmouseover`. In IE10, JavaScript programmers were granted the `classList` object, enabling us to add and remove classes without having to deal with tricky string manipulation.

And this is without touching on all of the amazing things that CSS can do without any JavaScript at all. CSS is finally getting powerful enough to truly facilitate the separation of the JavaScript logic from its presentation. In modern browsers, classes can represent the state things are in and the CSS can define what that means visually – but you can't do that if your code is littered with calls to `.hide()`.

And beyond all that, there are new, powerful APIs that you can't access through jQuery on its own: things like the new MutationObserver and WebRTC require you to talk to the browser itself. And, it turns out, that's not all that bad.

jQuery is a big, monolithic library. Having Ajax, DOM manipulation and all the rest in a single package locks us into a 10-year-old way of doing things

USE IT: DON'T DEPEND ON IT

We may not need jQuery any more, but what's the harm in using it if it makes your job easier? There isn't any! Keep using jQuery to build your sites and apps. But if you're building a library, consider not making it into a dependency.

Why? Because jQuery is a big, monolithic library. It includes Ajax, DOM manipulation and selection, events, promises, effects and a whole bunch of utilities. Having all of this in a single package locks us all into the jQuery way of doing things – which is also the 10-year-old way of doing things.

ALTERNATIVE SOLUTIONS

So what can we use instead? Q (*github.com/kriskowal/q*) does brilliant things with chaining promises and exception-handling. CSS animations (*daneden.github.io/animate.css*) are much smoother than anything you could do with JavaScript. Soon, custom elements (*netm.ag/custom-255*) are going to provide the best possible way of building reusable components. If we live in a world where you can pick and choose which DOM manipulation library you want, or whether you want to use RSVP.js (*github.com/tildeio/rsvp.js*) or Q for promises, we can end up with better open-source solutions to all of these problems.

Beyond all this, the jQuery plugin style puts every library you include in the same, ever-growing name pace. Everything is `$(el).colorPicker()`, or `$(el).chosen()`. This doesn't play nicely with a future in which we have real modules.

Modules will enable us to cleanly seperate our dependencies, and create optimised builds more easily. Whether you use Browserify, RequireJS or the modules added with ECMAScript 6, it's the future. A single, global namespace isn't.

So how do we reach this brave new world? The first step is for all the libraries we need to release their death grip on jQuery. If they become dependency-free, we can begin to imagine building a modern web app without it.

There is one important caveat: if you're targeting older browsers, including IE6 and 7 or Android 2.3, you aren't ready for this brave new world just yet. Enjoy jQuery to its limits and dream of a future when you can finally speak the browser's language.

PROFILE

Zack (@zackbloom) co-founded eager.io, and works to make open source tools accessible to all. He created *YouMightNotNeedjQuery.com* with Adam Schwartz (@adamfschwartz)

★JAVASCRIPT

ENHANCE, DON'T EXCLUDE

Progressive enhancement isn't about disabling JavaScript, no matter what its opponents claim. Build your sites for everyone, suggests **Westley Knight**

I've read many articles recently about why you should be building your web apps, sites, or whatever they may be, with JavaScript, and just JavaScript alone. Some people even go as far as to say that progressive enhancement is dead (see Tom Dale's article here: *netm.ag/tomdale-249*).

So what happens if you do build a site that way? Once the JavaScript is downloaded, everything comes to life. The layout and content loads in beautifully; the site feels amazingly responsive to the user; and the file size is tiny, oh so tiny. Look at how much bandwidth we saved by building everything in JavaScript! So what's the problem?

The problem is right at the start, right at the part that reads "once the JavaScript is downloaded". Those that claim progressive enhancement is on its way out seem to be forgetting that the foundation of the web is HTML. Without it, there is nothing for JavaScript to manipulate.

BUSTING MYTHS

Unfortunately, there seems to be a rather large misunderstanding of what progressive enhancement actually is. At An Event Apart Chicago, Clearleft's Jeremy Keith was quoted on Twitter (*netm.ag/keithquote-249*) as saying: "Progressive enhancement is avoiding a single point of failure".

In addition to this resonant statement, Jeremy went on to talk about the myth that progressive enhancement means designing for the lowest common denominator – when in fact, it means starting from there.

YOU ARE NOT YOUR USER

How many times, as a working designer or developer, have you had a problem that only arises as the client is using an older browser? I'm going to stick my neck out here and say 'countless times'.

How many times has a family member or friend asked for you to help sort out their computer, only to find out they aren't using your favourite browser, so you install it to try to convert them? Possibly a few.

And how many times have you watched a partially sighted user navigate a website using a screen reader and a keyboard, only to become frustrated at the simplest of tasks?

For half an hour, I saw such a partially sighted user struggle to do what should be

Who are we to pick and choose who can and cannot use what we build, if the World Wide Web is indeed intended for the whole world?

the simplest of things on an ecommerce site: order a product. Poorly implemented JavaScript caused them visible frustration. The dejection in their body language made me want to jump up and help.

But I'm not going to be there alongside every user when something happens that they don't expect, or when they can't find the information they're looking for. Neither are you.

Instead, what we can do is build the web to be as inclusive as possible: to not put up barriers between users and the information they're trying to get, or the task they are trying to complete.

ACCESSIBILITY FOR ALL

Sir Tim Berners-Lee, the inventor of the World Wide Web, once said: "Anyone who slaps a 'this page is best viewed with Browser X' label on a web page appears to be yearning for the bad old days, before the web, when you had very little chance of reading a document written on another computer, another word processor, or another network."

Today, we can see messages on sites along the lines of 'JavaScript must be enabled for this site to work'. This, to me, is far more worrying. At least when the page was best viewed in Browser X, anyone could read the content, even if it wasn't presented in the optimum format.

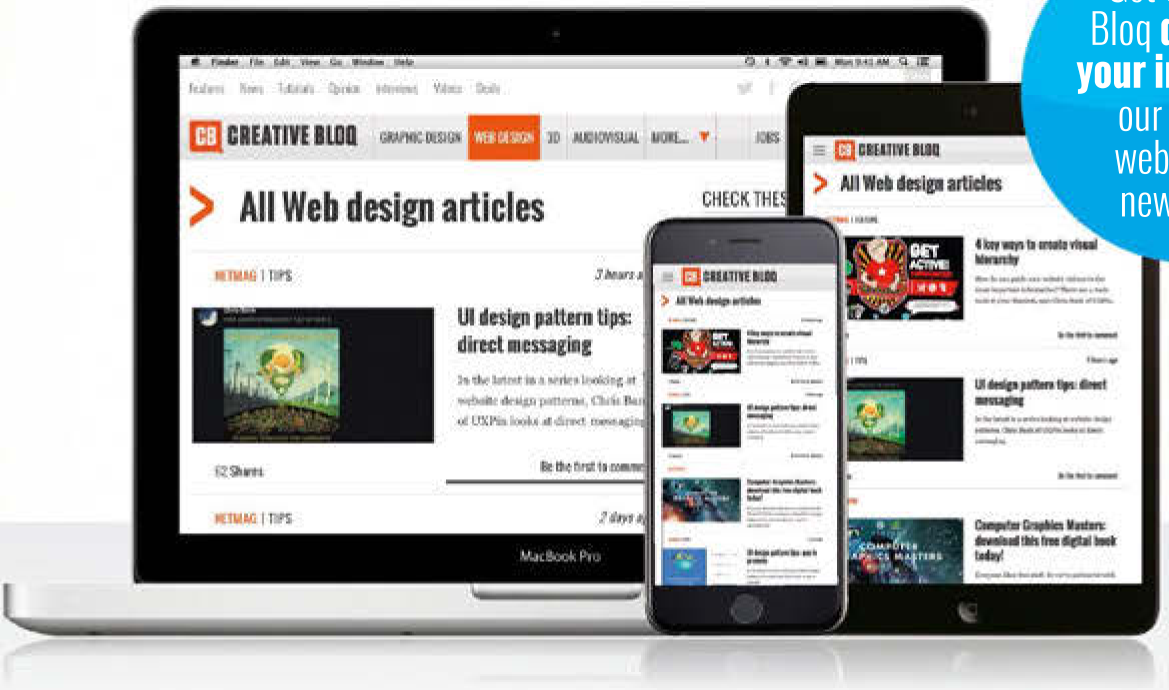
COURTESY TO OTHERS

I find it difficult to believe that people cannot consider the needs of others when building things for the web. It seems an extremely blinkered and closed-minded view – and the antithesis of the web itself.

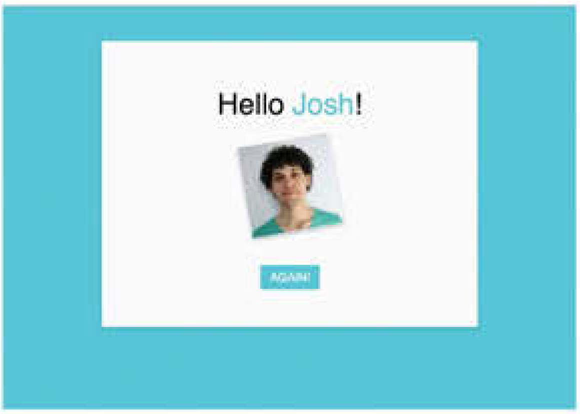
As Ethan Marcotte put it when responding to the article I referenced at the beginning of this column (*netm.ag/marcottequote-249*): "Progressive enhancement's not for those manually disabling JS. It's designing for resilience and reach."

The internet opens up an entire world of information to everyone with a connection to it, lets users perform complicated tasks at the click of a button, and, perhaps most importantly, enables people to connect with each other. Who are we to pick and choose who can and cannot use what we build, if the World Wide Web is indeed, as the name suggests, intended for the world? ◘

★ PROFILE
Westley (@westleyknight) is mobile UX architect at Next, love of frontend web, speaker, writer 'of sorts' and family man (*westleyknight.co.uk*)


CREATIVE BLOQ
GRAPHIC DESIGN
WEB DESIGN
3D
AUDIOVISUAL
MORE...
JOBS
All Web design articles
NETMAG | TIPS
UI design pattern tips: direct messaging
62 Shares
Be the first to comment
4 key ways to create visual hierarchy
Computer Graphics Masters: download this free digital book today!
MacBook Pro

APPS WITH FRAMEWORKS



Download the files here!

You'll find all the files you need for this tutorial at *netm.ag/ampersandGit-262*

ABOUT THE AUTHOR

KAMIL OGÓREK

w: *kamilogorek.pl*

t: @kamilogorek

job: Senior client-side engineer, X-Team

areas of expertise: JavaScript

q: how do you deal with stressful situations?

a: I go bouldering, lift heavy things or cook delicious food

Illustration: Lynn Fisher and Amy Lynn Taylor

AMPERSAND.JS

BUILD MODULAR APPS WITH AMPERSAND.JS

Kamil Ogórek introduces 'non-frameworky' framework Ampersand.js, and uses it to build an animated GIF aggregator

VIDEO

See the tutorial in action in Kamil Ogórek's exclusive accompanying video at *netm.ag/AmpersandVid-262*

Writing complex applications isn't an easy task. Modularisation – breaking things down into small pieces that work together – is one way to go. Ampersand.js is the new framework on the block, and it combines the best parts of the top JavaScript tools to help you do just this. Structurally, it's similar to Backbone, but it's all written in CommonJS modules, as fully separate pieces hosted on npm. All those pieces can be used together, and exposed to the world with Browserify.

In this tutorial, I am going to recreate one of the best things available on the internet right now – the animated GIF aggregator: *gifalicious.net*. This enables us to collect newly found images in local storage, to be used when needed.

STARTING OUR CODEBASE

For now, we'll keep things simple and use local storage, which means we don't need to create a local server to expose our files. We'll create our codebase using a few basic tools. For now, all we need is the folder structure, Browserify for working with CommonJS modules, Watchify (*github.com/substack/watchify*) so we'll be able to iterate more quickly, and of course our starting point – a single HTML file. Let's start by initialising our application using npm's `npm init`. This will take us through a few basic questions, so we can add in the application name, description, author, license and version.

Next we can install our tools with `npm install --save browserify watchify`, and add our `npm start` and `npm build` commands to the `package.json` scripts object, so we'll be able to start our development. We can add those commands with specific flags to our newly created `package.json`:

```
"scripts": {
  "build": "browserify js/app.js -o bundled.js",
  "start": "watchify js/app.js -o bundled.js"
}
```

We'll create a basic application structure that will hold our files, and write a simple index.html file:

```
<section data-hook="app-container"></section>
<script src="bundled.js"></script>
```

Ampersand.js is structurally similar to Backbone, but is written in CommonJS modules

APPLICATION LIFT-OFF

Now we have installed Watchify and configured our npm scripts, we can run these right away in the background with `npm start`. Now our bundled.js file has been created and will be recompiled every time we make a change.

Let's install some of the Ampersand.js pieces and write our first lines of code in our app.js file. What we'll need is `ampersand-state`, `ampersand-router` and `ampersand-view`.

```
var Me = require('./models/me');
var AppView = require('./views/app');
var Router = require('./router');
window.app = {
  init: function () {
    this.me = new Me();
    this.view = new AppView({
      el: document.body,
      model: this.me
    });
    this.router = new Router();
    this.router.history.start();
  }
```

▶

FOCUS ON

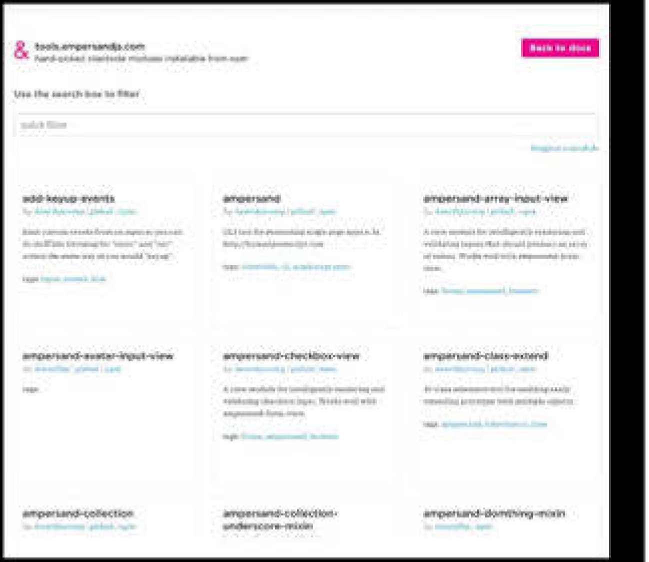
NODE PACKAGED MODULES

One of the most crucial things to consider when writing modularised applications is code reuse, and one of the simplest ways to achieve this is with npm. Node packaged modules (npm) hosts modules that have been written and published by other developers and can be used in your own application.

Thanks to npm, you can simply install the pieces of code you'd like to use and require them in the code. Some of those modules can be used on both client- and server-side in node.js apps, some are node.js only, and some are meant for browsers only.

To use those modules in your client-side application, you need to leverage the bundling capabilities of Browserify (written by James Halliday – @substack), which will transform all CommonJS-based requires into code that is usable in the browser.

Npm is public and you can make your own modules public, so you can use them whenever you need. Also, because npm is open source, you can host your own instances and create a local repository that contains all the common modules you need regularly, or all those that you're using on a particular project.



Modularised apps Ampersand Tools are npm hosted, hand-picked client-side modules, which can help you write your apps more quickly

IN-DEPTH

FURTHER READING

Although Ampersand.js is one of the frameworks that relies on good JavaScript knowledge rather than the internals of the framework itself, it's always a good idea to get to know your tools as well as possible. These resources will help you get up to speed with everything you need to make the best of Ampersand.js:

Ampersand.js homepage
ampersandjs.com
The Ampersand.js homepage is where you can find all the information regarding this framework, including documentation, how to contribute and how to get started.

Ampersand.js Learn page
ampersandjs.com/learn
The Learn page is the best way to start learning about Ampersand.js. This is the place you can read about basics of the framework and its ideology.

Human JavaScript
read.humanjavascript.com
This JavaScript book was written by Henrik Joreteg, the main creator of Ampersand.js, and covers solutions used in the framework.

Browserify handbook
github.com/substack/browserify-handbook
The Browserify handbook can help you grow your skills in using CommonJS modules. It covers plenty of common problems you can run into when developing applications using Browserify.

Being human Henrik Joreteg's book, *Human JavaScript*, explores practical patterns for JavaScript apps

```
};
window.app.init();
```

Let's look at what we've done here. We required our base state, `Me`. This is just a convention, but it contains all global application states. For now it'll just be a new, empty `AmpersandState` object.

FIRST MODULES

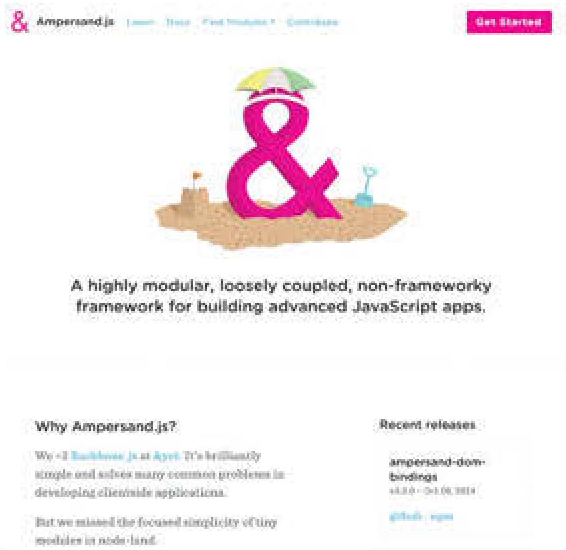
We have the main application view and our router, but right now we don't have those modules, so let's write them.

```
var AmpersandState = require('ampersand-state');
module.exports = AmpersandState.extend({});
```

The key thing to understand is that every time we use any of the Ampersand.js modules – for example `AmpersandState`, `AmpersandView`, Lo-Dash functions with an underscore namespace, or any constructor object that starts with a capital letter – it has been required at the very top of the file we're talking about. For more information on this, explore the Browserify website (*browserify.org*).

For now all our modules will be dead simple, as we're bootstrapping everything. Here you can see that we required `ampersand-state`. This is the base of all other modules, as it's capable of storing your data in a handy manner, reacting to events like add, change or remove, and can deal with changes of its own children if needed.

```
module.exports = AmpersandRouter.extend({
  routes: {
```



A highly modular, loosely coupled, non-frameworky framework for building advanced JavaScript apps.

Why Ampersand.js?

We <3 Backbone.js at &yet. It's brilliantly simple and solves many common problems in developing clientside applications.

But we missed the focused simplicity of tiny modules in node-land.

Recent releases

ampersand-dom-bindings
[illegible]

github · npm

Introducing Ampersand.js describes itself as a "highly modular, loosely coupled, non-frameworky framework for building advanced JavaScript apps"

```
    '': 'gifs',
    'gif/:id': 'gif',
    '*catch': 'catch'
  },
  gifs: function () {},
  gif: function (id) {},
  catch: function () { this.redirectTo(''); }
});
```

In router.js we're able to specify all of the application routes we'd like to handle, and eventually redirect users if they hit an unavailable route. Router is just an object of path/function key/value pairs, which will trigger a given function once hit.

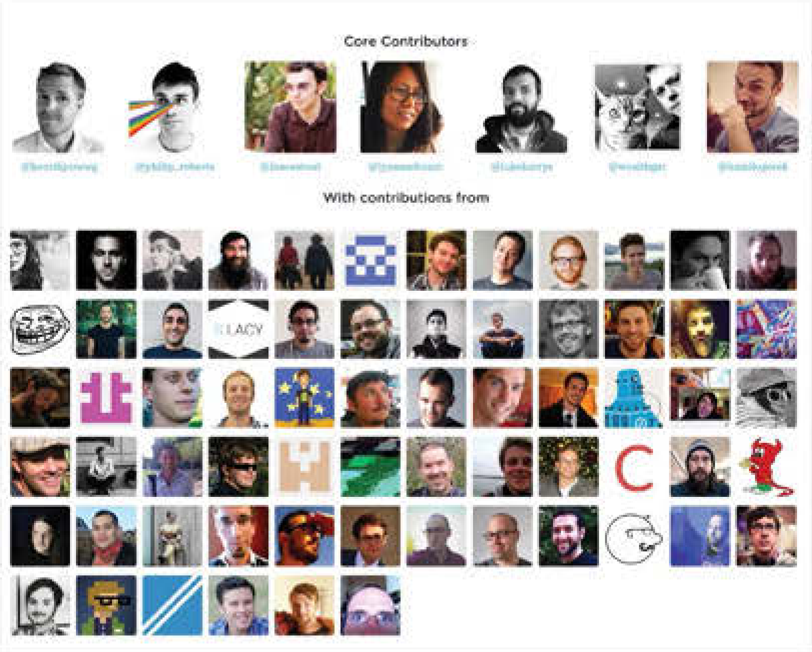
```
var _ = {
  result: require('lodash.result')
};
var AmpersandView = require('ampersand-view');
var AmpersandViewSwitcher = require('ampersand-view-switcher');
module.exports = AmpersandView.extend({
  initialize: function () {
```

In router.js, we're able to specify all the application routes we'd like to handle

```
    this.container = this.queryByHook('app-container');
    this.switcher = new AmpersandViewSwitcher(this.container, {
      show: function (newView) {
        document.title = _.result(newView, 'pageTitle') || 'Ampersand.js // Gifalicious';
        document.body.scrollTop = 0;
      }
    });
  }
});
```

Our main view is slightly more interesting. For this, we need to install three new dependencies. The `initialize` function is available on most of the Ampersand.js parts, and is called once a module has been created. This where we should attach listeners and events, and do our initial work.

What we do here is to query our element using the `queryByHook` function, which will look for an element with a given `data-hook` attribute. This way we're able to avoid using IDs, classes or attributes like `role`, allowing UI developers to modify them without breaking any logic.



Team effort Ampersand.js is a collaborative effort that is currently brought to you by over 70 contributors

Once we get this element, we'll use it to create our view switcher. This will let us swap its own content with anything we like, using its `set` method. In the app.js `init` function, we need to initialise our app state, create the main view passing `document.body` as its element, and set up routing.

HANDLING ROUTING

Because we don't have a server, we'll use hash-based routing. However, we're not able to serve the same file for every route, which is how you'd handle regular pushstate-based routing. There are three routes we have to handle: an empty route, which is our starting point, `gif/:id` route and `*catch`, which will simply redirect us to the starting point if there's no matching route.

Let's do the main route first. What should happen when user navigates here? We should create a new view, attach some data to it and display it in our view switcher.

```
app.view.switcher.set(new GifsView({
  model: app.me
}));
```

Now we have to create the view itself.

```
var AmpersandView = require('ampersand-view');
var GifsItemView = require('./gifs-item');
```

▶

RESOURCE

Ampersand.js: Minimalistic approach to not so minimalistic problems is Kamil Ogórek's presentation from 2014's Reject.js conference, in which he walks through the framework and explains the motivation behind the project: *netm.ag/reject-262*

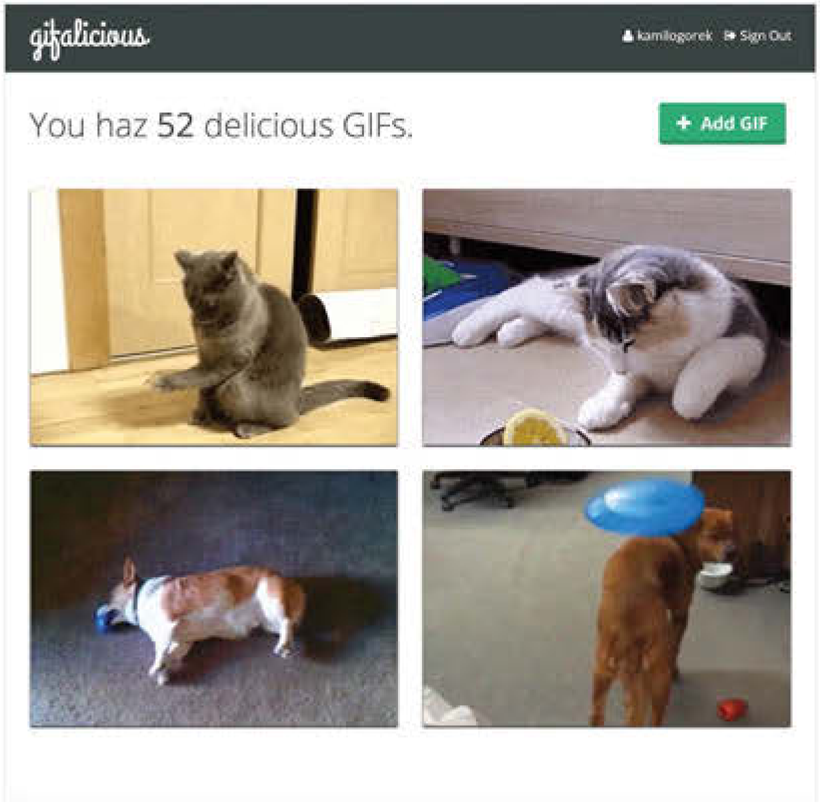
```
module.exports = AmpersandView.extend({
  template: require('../templates/gifs.html'),
  render: function () {
  this.renderWithTemplate();
  this.renderCollection(app.me.gifs, GifsItemView, this.queryByHook('gifs-container'));
  return this;
}
});
```

```
var AmpersandView = require('ampersand-view');

module.exports = AmpersandView.extend({
  template: require('../templates/gifs-item.html'),
});
```

ADDING NEW DATA USING FORMS

Because we don't have any initial data, we need to find a way of adding our precious GIFs to the local storage. We already have a collection which will handle all the storage stuff for us, so we only need to add a new model to this collection.

Let's add a form with a single input, which will take a valid URL and send it to our collection. First off, let's add a form tag with `add-gif` hook to the `gifs.html` template, and slightly modify our `views/gifs.js` to generate this form for us.



You haz 52 delicious GIFs.

GIFtastic The original animated GIF aggregator our creation is based on (*gifalicious.net*)

We need to introduce two new modules – `AmpersandFormView` and `AmpersandInputView` – plus some Lo-dash functions like `_.max` and `_.pluck`. We also need to add our `renderForm` function to the main `render`, directly under collection render.

```
renderForm: function () {
  this.form = new AmpersandFormView({
    el: this.queryByHook('add-gif'),
    submitCallback: function (obj) {
      app.me.gifs.add({
        id: app.me.gifs.length ? _.max(_.pluck(app.me.gifs.models, 'id')) + 1 : 1,
        src: obj.src
      });
    },
    fields: [new AmpersandInputView({ name: 'src' })]
  });
}
```

By creating forms this way, we're able to handle things like validation and requirements automatically. `AmpersandFormView`'s constructor takes an object with few options to configure its own behaviour. For our task we'll need three options:

By creating forms this way, we can handle things like validation automatically

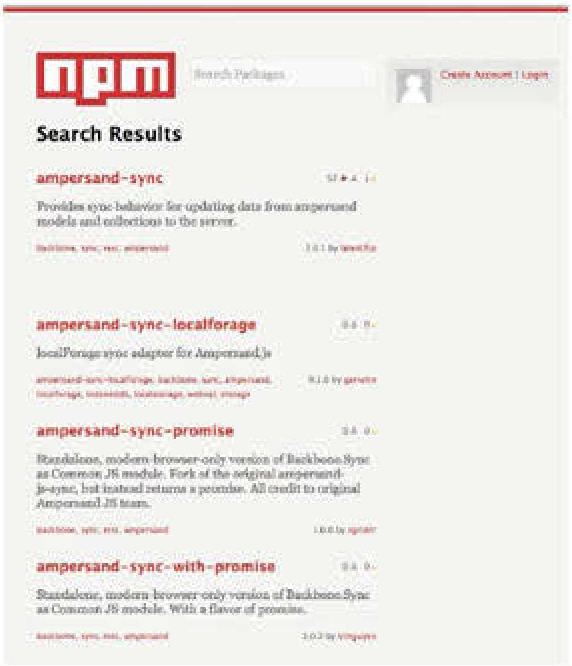
1. `el`, which is a DOM element that will be used as a container for our form
2. The `fields` array, to define what type of inputs should be rendered and included in the form
3. `submitCallback`, which is a function with a single argument: an object with all data that user has filled into the form

There are all kinds of inputs, which you can read about in the GitHub repo that goes with this tutorial (*netm.ag/ampersandGit-262*). Here you can see how to handle validation and duplicates, and take a look at a slightly more complicated example.

Now every time a user sends a new image URL, it will be added to our GIF collection, which will trigger the `writeToLocalStorage` function and store it there.

DISPLAYING DATA IN VIEWS

Now we have our view and data in place, we need to display it in our view. To do this, we'll use bindings,



Search Packages

Create Account | Login

Search Results

ampersand-sync

Provides sync behavior for updating data from ampersand models and collections to the server.

backbone, sync, rest, ampersand

ampersand-sync-localforage

localForage sync adapter for Ampersand.js

ampersand-sync-promise

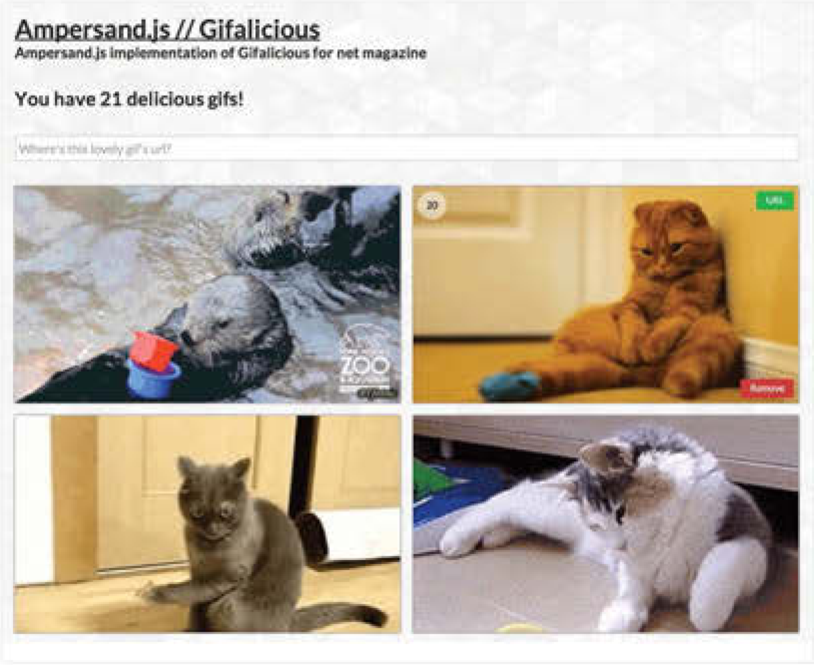
Standalone, modern-browser-only version of Backbone.Sync as Common JS module. Fork of the original ampersand-js-sync, but instead returns a promise. All credit to original Ampersand JS team.

backbone, sync, rest, ampersand

ampersand-sync-with-promise

Standalone, modern-browser-only version of Backbone.Sync as Common JS module. With a flavor of promise.

backbone, sync, rest, ampersand



Ampersand.js // Gifalicious

Ampersand.js implementation of Gifalicious for net magazine

You have 21 delicious gifs!

Where's this lovely gif's url?

Left npm lets us easily reuse components that have already been written by other people

Right The final form of our application. Isn't it cute?

which are a way to tell Ampersand what data should be displayed in our templates.

Bindings is one of the view's attributes that we can add:

```
bindings: {
  'model.id': {
    type: 'text',
    hook: 'id'
  },
  'model.url': {
    type: 'attribute',
    hook: 'src',
    name: 'src'
  }
}
```

Breaking things down a little, we're basically assigning specific behaviour to a given model's attribute, and we can set single or multiple bindings for the same attribute. First we define which attribute should be used (for example `model.id`), then specify its type (`type: text`) and select a hook (the DOM's `data-hook` attribute) of an element that should be used with our binding.

There are bindings like `class`, `text`, `attribute` and `booleanClass`. To learn more about all these possibilities, refer to the GitHub repo.

REMOVING GIFS

To get rid of data, we need to modify our `gifs-item` view and give it the ability to remove itself.

```
events: {
  'click [data-hook~=remove]': 'onRemoveClick'
},
onRemoveClick: function () {
  this.collection.remove(this.model);
}
```

The only new thing we're dealing with here are events. They work in exactly the same way as routes, but in this case, instead of paths, we specify an event and element on which it'll occur. For example, we're listening for a `click` event on an element with a `remove` `data-hook` attribute. When this happens, it will trigger the `onRemoveClick` handler and remove itself from our GIF collection.

FINISHING UP

The only thing left here is to add some CSS magic to make things a little prettier. To see the full GIF aggregator in action, go to *netm.ag/gifalicious-262*, where you'll find a fully functional demo. I've tried to keep things as simple as possible here. If you're interested and want to explore further, you should definitely check out the code repository I've provided – *netm.ag/ampersandGit-262* – for a more complicated version.

There are things like handling a single GIF route and adding a GIF counter which I've described in more depth in the project README file. There are plenty of other things we could do with our GIF aggregator – it's up to you where you take things from here! ◘

✓ PEER REVIEW

With thanks to Ryan Chartrand and Josh Johnston from X-Team, and &yet partner Henrik Joreteg for their peer reviews of this article.

Download the files here!
All the files you need for this tutorial can be found at *netm.ag/express-251*

ABOUT THE AUTHOR
SIMON MCMANUS
w: *simonmcmanus.com*
t: @simonmcmanus
areas of expertise: JavaScript, Node.js
q: what's the last photo you took on your phone?
a: a picture of a secret nuclear bunker. The one in Ongar

NODE.JS

WRITE MAINTAINABLE WEB APPS WITH EXPRESS

Simon McManus presents his guide to building web applications with Express, exploring modules, tools and techniques that'll make life easier

Express is a lightweight framework for building web applications with Node.js. Built on top of Connect (*senchalabs.org/connect*), it makes managing your web server, routes and middleware easy. Users include LearnBoost, LinkedIn, MySpace and Klout. It's also an unopinionated framework, so it doesn't tie you down to any particular architecture, templating engine, data store or authentication model. It's up to you to arrange your applications in a way that will scale and be easy to debug.

This tutorial will guide you through creating a sample application, using that as the basis for exploring some tools to keep your pieces small and loosely coupled.

To start off with, we're going to use the default web application provided by Express (*expressjs.com*). This article will assume you already have Node.js and npm (*nodejs.org*) installed. Let's install Express:

```
npm install express
```

To create a sample app, we type `express` followed by the application name. For the purposes of this tutorial, we're going to create a skeleton app. Let's call in skeleton:

```
express skeleton
```

First, we need to install our new apps dependencies by doing the following:

```
cd skeleton
npm install
```

To start the app, type:

```
node app.js
```

Once started, you can test the app by going to *http://localhost:3000*.

Tools like nodemon (*github.com/remy/nodemon*) make development much easier by restarting the Node process every time a file is changed. Let's install nodemon:

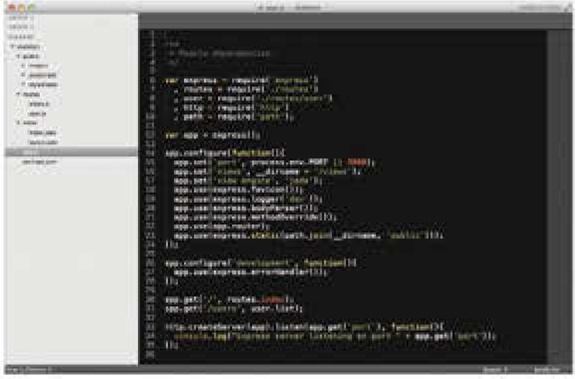
```
npm install -g nodemon
```

Then run our app with nodemon:

```
nodemon app.js
```

Now, as we make changes to our files, the application will be restarted.

The structure and style provided in the template Express app isn't to everyone's liking, and there's no



```
/**
 * Module dependencies.
 */

var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');

var app = express();

app.configure(function(){
  app.set('port', process.env.PORT || 3000);
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(path.join(__dirname, 'public')));
});

app.configure('development', function(){
  app.use(express.errorHandler());
});

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log("Express server listening on port " + app.get('port'));
});
```

Basic structure We'll just use the defaults in Express's basic structure

need to stick to it. However, there's a lot to be said for a consistent structure across your applications. When you've decided on your own structure or style, it's worth creating a GitHub repository containing a boilerplate application for your environment.

ROUTES

The sample `app.js` sets up various pieces of middleware and two routes, `/` and `/user`. Middleware and routes are the foundation on which most Express apps are built. They're the best place to start when trying to understand how an app is put together. When a request is received, Express first calls the middleware functions followed by the routes in the order specified until the requested URL matches a routing rule. Calling `next()` from within a middleware or route function passes the request on to the next middleware/route handler. It's for that reason we declare our most specific URL routes first and most generic routes last. The Express API is very powerful and time spent reading the documentation (*expressjs.com/api.html*) is never wasted.

> It's worth making an effort to prevent becoming too tightly coupled with Express

It's worth making an effort to prevent becoming too tightly coupled with Express, or any other framework for that matter. Try to ensure your functions and modules have no knowledge of the Express interface. This also makes the code much easier to test and reuse.

COMMONJS

Node.js uses CommonJS (*wiki.commonjs.org/wiki/CommonJS*) to require files, so it's useful to

FOCUS ON

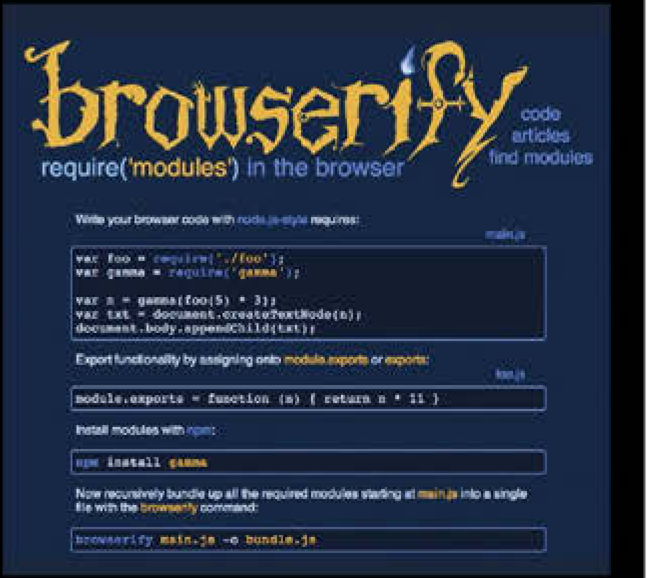
REUSABLE MODULES

Always try to write small reusable modules. Make your modules do one thing and do that one thing very well. Breaking your app down into reusable modules will help with debugging, scaling and maintenance in the long term. There are tools to help you. npm (*npmjs.org*) and package.json (*package.json.nodejitsu.com*) files will make your life easier. Understand their potential. Use them wisely.

Bear in mind tools like Browserify (*browserify.org*), which make it easy to use CommonJS and npm modules in your browser code. Keeping client-side code in npm is one thing, but you can take it a step further by writing isomorphic modules. Isomorphic JavaScript can run on the client and the server. In an AJAX application an obvious use case is data validation.

Once you've got your head round routing on the server, you also may want to think about how you could reuse routes in your client-side code. Page.js (*visionmedia.github.io/page.js*) is a 'Micro client-side router inspired by the Express router'.

Written by TJ Holowaychuk (also the author of Express), it uses the same routing syntax so can provide some interesting opportunities to share routes between the browser and your Node.js application.



Write your browser code with node.js-style requires:

main.js

```
var foo = require('./foo');
var gamma = require('gamma');

var n = gamma(foo(5) * 3);
var txt = document.createTextNode(n);
document.body.appendChild(txt);
```

Export functionality by assigning onto module.exports or exports:

foo.js

```
module.exports = function (n) { return n * 11 }
```

Install modules with npm:

```
npm install gamma
```

Now recursively bundle up all the required modules starting at main.js into a single file with the browserify command:

```
browserify main.js -o bundle.js
```

Easy modules Browserify makes it easier to use modules in browser code

RESOURCES

FURTHER READING

Node.js (*nodejs.org*) is a platform built on Chrome's JavaScript runtime for building fast, scalable network apps. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time apps that run across distributed devices.

Express guide
Express (*expressjs.com/guide.html*) is a minimal and flexible node.js web application framework, providing a robust set of features for building single and multi-page, and hybrid web applications. Also see the Express API (*expressjs.com/api.html*).

Package.json
When we set up the Express app, it created a package.json (*npmjs.org/doc/json.html*) file that was used to install its dependencies. It specifies the packages required by your application. See the package.json interactive guide: *package.json.nodejitsu.com.*

Node.js Debugger
The manual and documentation for Debugger (*netm.ag/debug-251*).

DailyJS
Some interesting new flow control libraries and information about CommonJS modules from DailyJS (*dailyjs.com/2012/02/20/new-flow-control-libraries, dailyjs.com/2011/11/14/popular-control-flow, dailyjs.com/2010/10/18/modules*).

Scaling Isomorphic JavaScript
Scaling isomorphic JavaScript code on the nodejitsu (*blog.nodejitsu.com/scaling-isomorphic-javascript-code*).

Also try these debugging tools:
Nodetime (*nodetime.com*), node-profiler (*netm.ag/profiler-251*), node-webkit-agent (*netm.ag/agent-251*) and node-heapdump (*netm.ag/heap-251*).

▶ understand what's possible with CommonJS. In the example app.js we can see the line:

```
routes = require('./routes')
```

As `/routes/` is a folder, it looks for an `index.js` file and returns anything on the exports object in that file. The same thing happens with the `routes/user.js` file; it adds a list function to the exports object. When required in `app.js` the list function is available on the returned object. For example:

```
require('./routes/list.js').list()
```

If you want your module to expose a single function you can do:

```
module.exports = function() {}
```

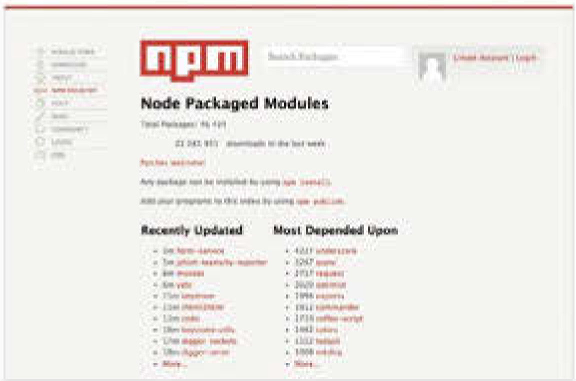
USING NPM

At the time of writing, the public npm repository holds over 50,000 packages, which you can browse at *npmjs.org* and install with the command we used earlier. npm is an open platform so you should look over code before using it in your production applications. Once you've got the hang of using other people's modules, try publishing some of your own code to the npm repository, which will require a package.json file.

PACKAGE.JSON

When we set up the Express application, it created a package.json file that was used to install its dependencies. It specifies the packages required by your application. You saw them being installed when you ran the npm install command.

It's possible to specify exactly which version of each module should be used. If you don't care which version is used, you can use the wildcard (*) but that leaves you at the mercy of the plugin author. If your package author uses semantic versioning, you could



Package manager npm installs, publishes and manages Node programs specify the major and minor version, but always get the latest patches: 1.2.x.

Even this can be risky as not everyone uses semantic versioning properly. When it comes to creating testing and production builds you should look at the npm shrink-wrap command. It creates an `npm-shrinkwrap.json` file that specifies exactly which version of your dependencies (and their dependencies) should be used.

It's worth understanding what else is possible with a package.json file. One example is specifying commands to run tests (*npmjs.org/doc/cli/npm-test.html*) and start (*npmjs.org/doc/cli/npm-start.html*) your application. This comes in handy when working across different projects. Regardless of which test framework is used, the same command will run the test suite for any given project.

FLOW CONTROL

It's an accepted convention in Node that the signature of a callback function is `function(error, data)`. It's worth following this convention wherever possible. Especially when it comes to using flow control libraries. Writing code with lots of callbacks can quickly become difficult to manage. Flow control libraries help to solve this problem. Async (*github.com/caolan/async*) seems to be one of the must established, but there are alternatives (*dailyjs.com/2011/11/14/popular-control-flow/*). It's really useful to become familiar with at least one of these flow libraries.

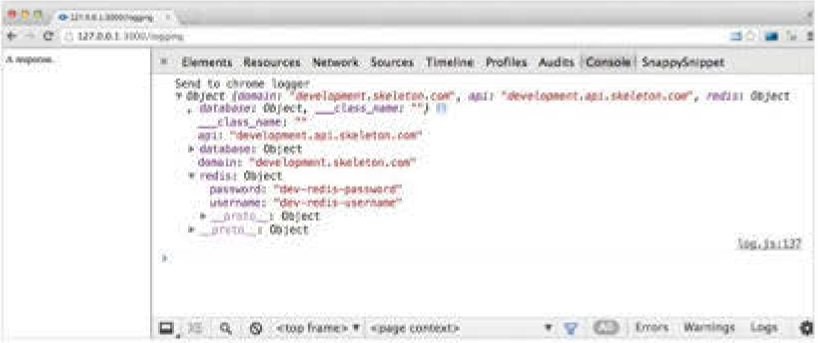
Node.js (particularly npm) has a massive ecosystem that's constantly changing

CONFIGURATION

At some point, when building your application, you're going to want a different configuration depending on the environment. In simple cases, this can be done using an environment variable. We can see this in app.js where it uses the variable: `process.env.PORT`. In more complicated situations, there are modules such as `config` (*github.com/lorenwest/node-config*), which make it a lot easier to work with multiple configurations.

Using the config module is simple. First you need to install it:

```
npm install config -save
```



```
Send to chrome logger
▼ Object {domain: "development.skeleton.com", api: "development.api.skeleton.com", redis: Object
  , database: Object, __class_name: ""}
    __class_name: ""
    api: "development.api.skeleton.com"
  ▶ database: Object
    domain: "development.skeleton.com"
  ▼ redis: Object
      password: "dev-redis-password"
      username: "dev-redis-username"
    ▶ __proto__: Object
  ▶ __proto__: Object
                                                                                      log.js:137
```

Chrome logger Chrome Logger allows you to send logs from your node application to the Chrome browser where it can be more easily interrogated

The `-save` flag will update the package JSON with the package you're installing. Once installed, you need to create a folder in the root of your project called `config`. In there you can create files for each environment. For example, `default.json`, `development.json` and `production.json`. You can use JS, JSON or YAML files for your configurations. When you've created the config files, you just need to require the config module and that will load the config appropriate to your current environment:

```
var CONFIG = require('config')
```

RESOURCE

EXPRESS GUIDE

To help you through the basics of building an Express application in a little more detail than this article is able to, try Express Guide: *expressjs.com/guide*

DEBUGGING

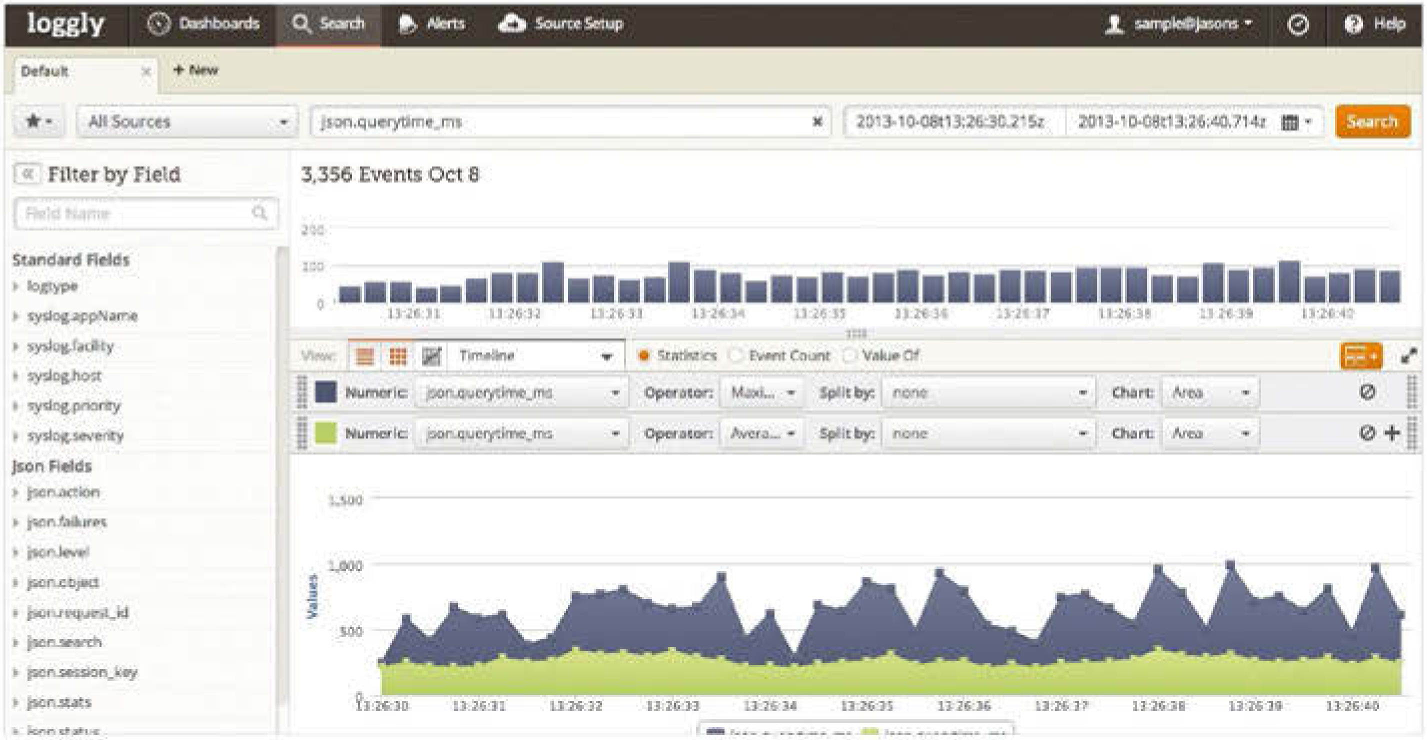
There are lots of debugging tools available. The subject is worthy of an article in its own right, but here is some info about the basic tools. You can run Node.js in debug mode, which allows you to add breakpoints and debugging statements. You can even hook it up to the blink developer tools (formally WebKit Web Inspector) with Node Inspector (*github.com/node-inspector/node-inspector*).

This is often overkill if you just want to inspect a variable. Sometimes a `console.log` is all you need. They work the same as in the browser, but sends output to the Node.js console. When inspecting large objects, this is not always ideal as it dumps out the whole JSON object as a string. In such a situation, Chrome Logger (*craig.is/writing/chrome-logger*) can come in handy. Chrome Logger has implementations available for many programming languages and allows you to log objects from your server side code to the Chrome console. In our example app we just need to add a piece of middleware:

```
app.use(require("express-chrome-logger"));
```

You then need to install the Chrome Logger Extension (*netm.ag/chromelogger*) and turn it on by clicking its icon so it turns blue. From within a route you can then call:

►



Log management Cloud-based log management services like *loggly.com* collect log files from different servers. Used with JSON logging, it's simple to fliter your logs by the fields you specify

```
res.console.log('Send routes to chrome logger', CONFIG);
```

You'll see the results in your Chrome console.

JSON LOGGERS

Bunyan (*github.com/trentm/node-bunyan*) is a JSON logging library that allows you to log at different levels and then filter the results:

```
node app.js | bunyan -l error
```

That command will run our Node.js app, sending the results to Bunyan, which will filter the output and only show errors. This JSON logging makes it easier to extract and analyse information from your logs.

STYLE GUIDES AND CONSISTENCY

When maintaining a codebase, it's important to use the style and conventions already in place. Style guides are a good way to share these convention. Two good examples are JavaScript Style Guide (*bengourley.co.uk/javascript-style*) and Node.js Style Guide (*netm.ag/styleguide-BZ11*). I particularly like how Ben Gourley explains the thinking behind the points in the first example.

To take it a step further, you can introduce a `.jshintrc` file to your projects. When used with Sublime (*sublimetext.com*) and Sublime Linter (*github.com/SublimeLinter/SublimeLinter*), or any other good JavaScript editor, it will highlight when you don't follow the JSHint rules specified.

SUMMARY

Use the tools described in this article to help keep your modules small and loosely coupled. Where possible, write completely separate applications. With such an arrangement, it'll be easier to swap out components when necessary. It'll also make testing and scaling much easier. Remember that Node.js (and particularly npm) has a massive ecosystem that's constantly changing, so it's worth Googling any problems you may encounter.

There's a basic application showing some of the techniques and tools from this article available at *netm.ag/express-251*. As applications go, it's not particularly useful, but it may help you with working out how to use some of the tools discussed.

As always, tools can only do so much. It's important to create good README.md files, write tests, include code comments and, wherever possible, keep things simple. The Code for Maintainers wiki (*c2.com/cgi/wiki?CodeForTheMaintainer)* says it best:

"Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live. Alternatively, always code and comment in such a way that if someone a few notches junior picks up the code, they will take pleasure in reading and learning from it."

VIDEO

Watch an exclusive screencast of this tutorial created by the author at *netm.ag/tut4-251*

ABOUT THE AUTHOR

ALEX MATCHNEER

w: *machty.com*

t: @machty

job: Lead engineer, Express Checkout; Core team member, Ember.js

areas of expertise: Client-side JS applications

q: what was your childhood nickname?
a: Little Man

HEAD TO HEAD

REACT VS EMBER.JS

Alex Matchneer takes a look at the differences between React and Ember.js, and considers which to use and when

	REACT	EMBER.JS
	An open source, Facebook-sponsored JavaScript library for building user interfaces. It ditches traditional data binding in favour of a unidirectional, 're-render everything' approach that is component-centric.	An open source, heavyweight framework for building client-side web applications. It provides universal data binding and a URL-driven approach to structuring applications with a focus on scalability.
SCOPE	React's homepage describes it as "a JavaScript library for building user interfaces" (*netm.ag/react-260*). React provides a simple yet performant API for client-side rendering, and can be used as the rendering engine for other JavaScript libraries (including Ember).	Ember.js' homepage describes it as "a framework for creating ambitious web applications" (*emberjs.com*). Client-side rendering is one of the features Ember provides, but it also intends to provide structure to scalable web applications beyond just the view layer.
MANAGING STATE	React implements a unidirectional data flow, so whenever a component's `setState` method is called, that component (and any of its children) will re-render. Data flows downwards, events (clicks, form submits) flow upwards and ultimately translate into a `setState`.	Ember's object model facilitates Key-Value Observation, which allows Ember to re-render changes to specific properties. Two-way binding is supported, but for the most part, Ember encourages unidirectional data flow.
SCALABILITY	React is arguably the most performant and flexible rendering library around, even when rendering large lists. That said, it doesn't intend to provide any sort of application structure, and must be used in conjunction with other libraries or architectures (such as Flux, react-router) for any medium- or large-scale apps.	Ember's patterns for structuring applications are baked into the framework in a convention-over-configuration manner and have seen great success in the ecosystem of large JavaScript applications. That said, Ember's view layer is less performant when rendering large lists, particularly relative to React.
SECRET WEAPON	Every time `setState` is called, React generates a new virtual DOM with the latest data and efficiently diffs it against the previous version, generating a minimal list of changes that need to be made to the real DOM to bring it in sync.	Ember's recent secret weapon is `ember-cli`, the official command line interface for building and managing Ember applications. `ember-cli` makes it incredibly simple to generate and import Ember addons into your projects.

VERDICT

Keeping in mind that React is a UI library and Ember is an application framework, you should give both React and Ember a try. If you're building something large, lean towards Ember. If you're building something smaller but performance-sensitive, opt for React.

FACT FILE

DATA-BINDING

Ember features one- and two-way data-binding via Key-Value Observation (KVO). React doesn't use KVO, but instead re-renders using a unidirectional data flow originating from calls to `setState`.

CONTRIBUTORS

React was built and is actively maintained by Facebook. It was open-sourced in mid-2013. Ember was forked from SproutCore in 2011 and is maintained entirely by the Ember community.

View source files here!
You'll find all the files you need for this tutorial at *netm.ag/polymer-264*

ABOUT THE AUTHOR
ROB DODSON
w: *robdodson.me*
t: @rob_dodson
job: Developer advocate, Google
areas of expertise: Web development, HTML, CSS and JS, Web Components
q: what's the first gadget you owned?
a: An Optimus Prime

WEB COMPONENTS

BUILD AN APP USING WEB COMPONENTS

Rob Dodson explains how to build your first application by using the Polymer library to leverage the power of Web Components

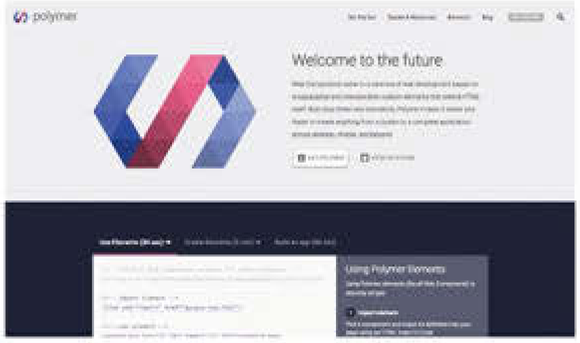
Web Components are an amazing set of new standards that enable developers to extend the HTML vocabulary by creating their own elements. This simplifies the work required to put together an application, because you can leverage third party components like toolbars and sliding drawer panels, without copying and pasting tons of markup into your code base. Although Web Components may seem like a futuristic technology, we can actually start using them today, thanks to a set of polyfills called webcomponents.js (*webcomponents.org*) and libraries like Polymer (*polymer-project.org*). Polymer adds syntactic sugar to the Web Components APIs to make them easier to work with. This sugar includes things like data bindings, attribute change watchers and a nice declarative syntax for creating your own elements.

Polymer also includes sets of pre-built elements called Core elements and Paper elements. Leveraging pre-existing elements means we can bolt together an application in no time. Let's dive in!

GETTING STARTED

First, make sure you've installed the components using Bower (see 'Installing components' in the boxout opposite for more on how to do this). You'll also need to make sure that you're previewing your application using a local server. You can do this by

VIDEO
Rob Dodson has created an exclusive screencast to go with this tutorial. Watch along at *netm.ag/polymervid-264*



Introducing Polymer Polymer makes it easy to create components

setting up Apache, node.js, or even using this helpful Python command: `python -m SimpleHTTPServer`.

From the supporting files for the tutorial, open index.html and add the Web Components polyfills where it says `<!-- place polyfills here -->`:

```
<script src="bower_components/webcomponentsjs/
webcomponents.js"></script>
```

Congratulations, you just catapulted your app into the future!

Immediately after the polyfills, include a `<link>` tag and set the `rel` to `import` and the `href` to `elements.html`. You may be familiar with using link tags to pull in style sheets, but this link tag is special. It's an HTML Import, one of the new technologies afforded to us by Web Components, and it allows us to load an entire document full of resources. Pop open elements.html and you'll see that there are even more imports, pulling in all of the components we'll use in our application.

> Polymer adds syntactic sugar to make the Web Components APIs easier to work with

COMPOSING ELEMENTS

With all of our elements imported, we're ready to start composing them on the page. Let's begin with the `<core-toolbar>`, a simple container that lays out its children horizontally using CSS Flexbox.

Add the following markup where it says `<!-- place components here -->`:

```
<core-toolbar>
 <div>My App</div>
</core-toolbar>
```

FOCUS ON

INSTALLING COMPONENTS

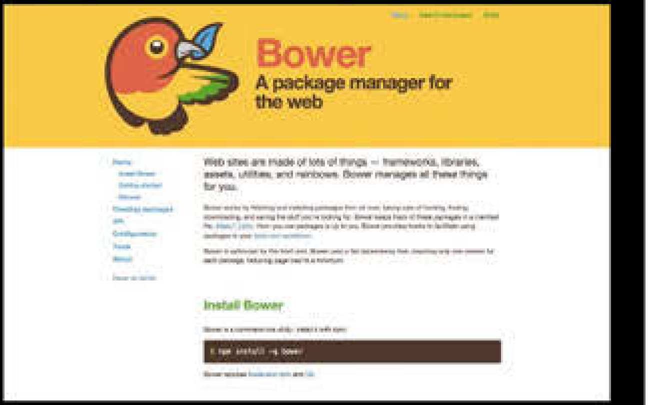
Before we can start building our application, we'll need to download the Core and Paper element sets. The easiest way to do this is with a package manager like Bower. If you're new to bower, take a gander at the website (*bower.io*).

Create a new file called `bower.json` and add `core-elements` and paper-elements to the dependencies object:

```
{
 "name": "netmag-polymer-app",
 "version": "0.0.0",
 "dependencies": {
  "core-elements": "Polymer/core-elements#^0.5.1",
  "paper-elements": "Polymer/paper-elements#^0.5.1"
 }
}
```

From your command line, cd into the same directory as your bower.json and run `bower install`. Bower will whisk away to GitHub to find the element sets and download them to your machine in a directory called 'bower_components'. When all is done, pop open that directory and take a look at all of the components you now have access to.

Not only did Bower grab the `core-*` and `paper-*` elements, it also grabbed their dependencies, including Polymer and the webcomponents.js polyfills. We now have everything we need to start composing our first app.



Web sites are made of lots of things — frameworks, libraries, assets, utilities, and rainbows. Bower manages all these things for you.

Installing elements Package manager Bower provides a convenient way to install the element sets you need

▶ And just like that, we have our toolbar! Because components compose so well, let's also drop in a `<paper-icon-button>`:

```
<core-toolbar>
 <paper-icon-button icon="menu"></paper-icon-button>
 <div>My app</div>
</core-toolbar>
```

Notice that the icon button and the title are laid out horizontally. Even though the title is contained within a `<div>`, it doesn't wrap (this is CSS Flexbox in action). You may also notice there's a default margin applied to our body, which creates a grey border around the entire application. We can get rid of this by applying some CSS to the body, or we can take advantage of Polymer's layout attributes.

These enable us to easily apply common CSS patterns to our markup. To remove the margin on our body, we'll give it a fullbleed attribute:

```
<body fullbleed>
```

> Web Components are a big change, bringing a ton of opportunities to do amazing things

If you take a peek in the dev tools you'll see that the fullbleed attribute applies a `margin: 0` and `height: 100vh` style to the body.

Next, wrap your toolbar in a template, like so:

```
<template is="auto-binding" id="app">
 <core-toolbar>
  <paper-icon-button icon="menu"></paper-icon-button>
  <div>{{headline}}</div>
 </core-toolbar>
</template>
```

The auto-binding template is a custom element that extends the native `template` tag (another new Web Components feature) and imbues it with the power of data binding. An auto-binding template will immediately stamp out its content and, if it finds a data binding, attempt to resolve the value.

To populate the value for `{{headline}}`, add the following JavaScript:

```
<script>
 var app = document.getElementById('app');
 app.addEventListener('template-bound', function() {
```

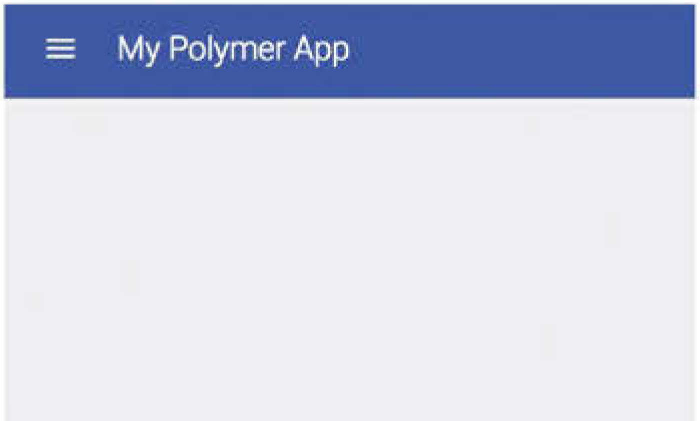
★ RESOURCES

FURTHER READING

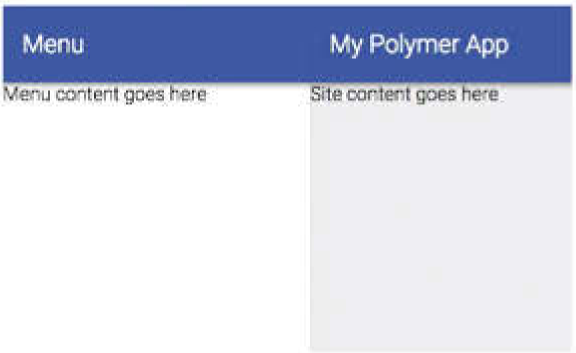
This article only scratches the surface of what's possible with Web Components. If you're curious to know how to take this app further by adding routing and keyboard accessibility, take a look at this guide on the Polymer site by Eric Bidelman.
polymer-project.org/articles/spa.html

If you'd like to know how to create your own elements, the Polymer site has some really nice starter tutorials. These will walk you through every step of the process, from creating a simple element all the way to building your own (mini) social network.
polymer-project.org/docs/start/creatingelements.html

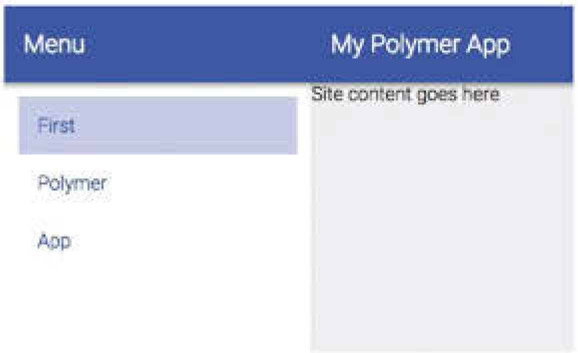
Lastly, I would be remiss if I didn't mention our bi-weekly YouTube series, Polycasts, which explains the many elements in the component ecosystem in small, bite-sized chunks. Take it for a spin and leave a comment to let us know what kind of shows you'd like to see in future.
netm.ag/polycasts-264



Starting out Fullscreen components in all their glory! It's a start, at any rate



Drawer Underlying structure <core-drawer-panel> gives a solid foundation



Menu Basic menu <core-menu> highlights the menu item selected

SLIDEDECK

See the slides from Rob Dodson's 'Let's build some apps with Polymer' presentation at *netm.ag/polymer2-264*

```
    app.headline = 'My Polymer App';
  });
</script>
```

Refresh the page and you should see the words 'My Polymer App' on screen.

It's a modest beginning, but at this point we have everything we're going to need to wire up the rest of our application. Let's see if we can turn this into a full-blown responsive scaffold.

GOING RESPONSIVE

Replace the markup in your template to match the following snippet:

```
<core-drawer-panel>
  <core-header-panel drawer>
    <core-toolbar>
      <div>Menu</div>
    </core-toolbar>
    <div>Menu content goes here</div>
  </core-header-panel>
  <core-header-panel main>
    <core-toolbar>
      <paper-icon-button icon="menu"></paper-icon-button>
      <div>{{headline}}</div>
    </core-toolbar>
    <div>Site content goes here</div>
  </core-header-panel>
</core-drawer-panel>
```

Refresh the page, and things should start to look a little more app-like.

The `<core-header-panel>` is a container that manages the relationship between a `<core-toolbar>` and any content that surrounds it. Try replacing the text that says 'Site content goes here' with several paragraphs until a scrollbar appears. Notice that if you scroll the page, the `<core-toolbar>` remains pinned at the top. That's the `<core-header-panel>` telling our `<core-toolbar>` to have a sticky behaviour.

The `<core-drawer-panel>` is a responsive scaffold that takes any child with a `drawer` attribute and places it into the side drawer. If it sees a child with a `main` attribute it places it into the primary content area. Try resizing your browser window down and you'll see that the drawer area moves out of the way when the screen size is too small. That's because the `<core-drawer-panel>` is responsive by default.

You may have noticed that clicking the menu button doesn't actually cause the drawer to slide out. We can fix that! Update your `<paper-icon-button>` so it has a `core-drawer-toggle` attribute. Now the button will trigger the drawer to slide out, and on a larger screen the menu button will be hidden.

At this point we have the shell of an application, but there's still no way to change from one section to the next. Let's give our app a menu so it's easier to navigate. Find the `<core-header-panel>` with a drawer attribute. Replace the `<div>` after the `<core-toolbar>` with the following markup:

```
<core-menu selected="0">
  <paper-item noink label="First">First</paper-item>
  <paper-item noink label="Polymer">Polymer</paper-item>
  <paper-item noink label="App">App</paper-item>
</core-menu>
```

Now when you refresh the page you should see a vertical menu in the drawer, with the first item highlighted.

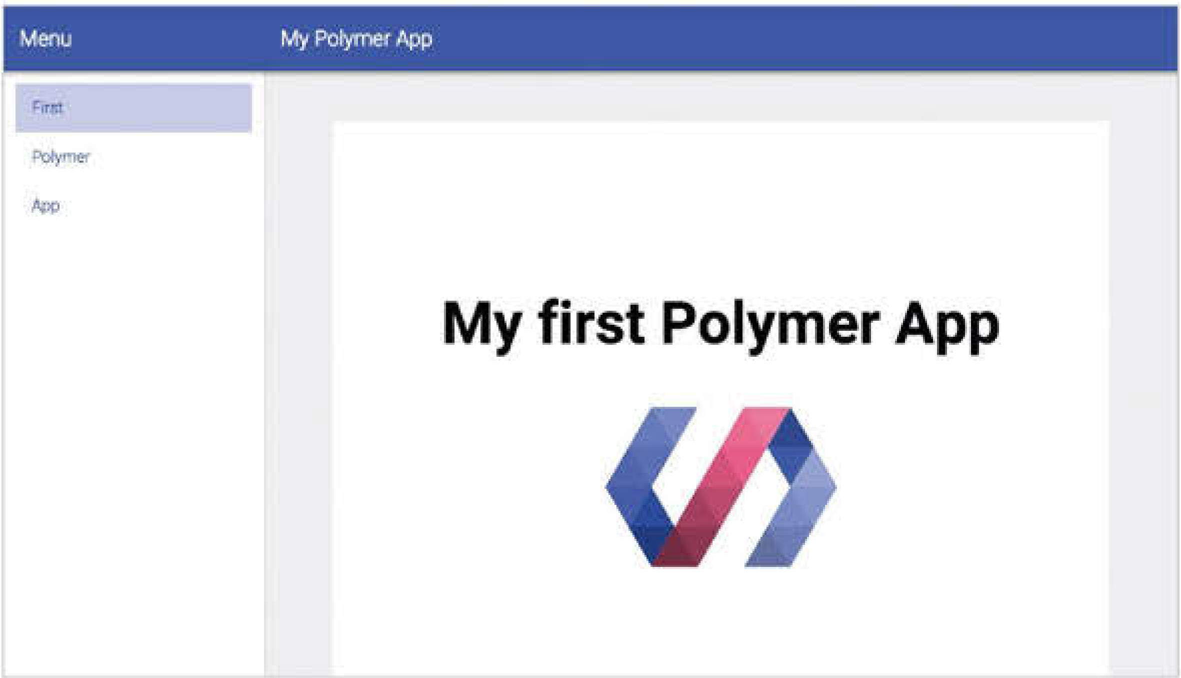
One useful feature of the `<core-menu>` is that it can select a child based on an attribute value. Update the first line of your `<core-menu>` so it looks like the following markup:

```
<core-menu selected="App" valueattr="label">
```

Notice that the third menu item (the one with the matching `label` attribute) is now highlighted. Because we're inside an auto-binding template, we can use data bindings to make our menu more

▶

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL



Final design The finished app <core-pages> makes it a breeze to swap content

dynamic. Update the `<core-menu>` one last time and replace the selected value with a binding for `{{page}}`:

```
<core-menu selected="{{page}}" valueattr="label">
```

Inside the same script where you set your headline value, set an additional value for the selected page:

```
app.addEventListener('template-bound', function() {
 app.headline = 'My Polymer App';
 app.page = 'First';
});
```

And for fun, update the `<div>` that says 'Site content goes here' with a `{{page}}` binding as well:

```
<core-header-panel main>
 ...
 <div>{{page}}</div>
</core-header-panel>
```

Now clicking on a menu item will change the content of the page. Nice!

SWAPPING BETWEEN SECTIONS

To wrap things up, let's create a few different sections for our app and swap between them when a menu item is clicked. To do this, we'll replace that last `<div>` with a `<core-pages>` element. The typical layout for a `<core-pages>` element looks like this:

```
<core-pages selected="0">
 <section>...</section>
 <section>...</section>
 <section>...</section>
</core-pages>
```

Changing the selected attribute on the `<core-pages>` element will update whichever `<section>` is currently being displayed. Since `<core-pages>` and `<core-menu>` both inherit from a common ancestor called `<core-selector>`, their APIs are nearly identical.

That means that we can hook up our `<core-pages>` with data bindings, just like we did with our `<core-menu>`:

```
<core-pages selected="{{page}}" valueattr="label">
 <section label="First">...</section>
 <section label="Polymer">...</section>
 <section label="App">...</section>
</core-pages>
```

Now you can place whatever content you want inside of your `<section>` elements to toggle between the different states by clicking a button. Congratulations, you've just built your first Polymer app!

Web Components are a big change for the platform, and with them come a ton of opportunities to do amazing new things. We covered a lot of ground in this walkthrough, but we've still only just scratched the surface of what's possible. The 'Further reading' boxout lists resources to help you explore Web Components in more detail. Keep hacking, and together we can componentise the web!

VIDEO

Watch Rob Dodson's video walkthrough on setting up Bower and Polymer at *netm.ag/bowerpolymer-264*

HEAD TO HEAD

COFFEESCRIPT VS TYPESCRIPT

Joe Maddalone scrutinises JavaScript compiling languages CoffeeScript and TypeScript, and compares their pros and cons

ABOUT THE AUTHOR

JOE MADDALONE

w: *joemaddalone.com*

t: @joemaddalone

areas of expertise:
JavaScript developer, instructor at egghead.io

COFFEESCRIPT	TYPESCRIPT
Created in 2009 by Backbone and Underscore author Jeremy Ashkenas.	Released in 2012 from a Microsoft team led by Anders Hejlsberg, creator of C#, Delphi and Turbo Pascal.
SOLVING JAVASCRIPT PAINS	
CoffeeScript introduces a new Python-like syntax favouring indentation over braces or parentheses. It expresses complicated prototypal inheritance with a lot less ceremony than TypeScript. Add to that eliminating `var` and `function` and the result is writing much less code.	TypeScript implements ES6 features alongside modules and interfaces, but not a new syntax. Its approach is more focused on tooling and analysis than writing less code. This results in early and specific error detection and a familiar syntax, and there's also excellent IDE integration.
IN THE WILD	
Depending on who is crunching the numbers, it is one of the top 15 languages being used on GitHub today, and job boards are flooded with requests for CoffeeScript developers.	Despite being a well-designed language with vast potential for organising complex applications, adoption has been slow – most likely owing to its mere association with the monolithic Microsoft.
CURRENT STATE OF THINGS	
Currently on version 1.8.0 and has 167 contributors – although the majority of contributions come from only a few of them.	Currently at version 1.4. The project has a team of 33 contributors made up almost exclusively of Microsoft employees.
IMPACT	
Despite having an entirely 'un-JavaScript' syntax, CoffeeScript seems to be shaping the future of JavaScript, and many of its core features are now in the ES6 specification.	The tightly structured implementation of TypeScript provides Visual Studio developers with unmatched tooling and management and, as a by-product, greater productivity.

VERDICT

It could be said that both these languages are doomed once ES6 arrives. But that would be short sighted, at least for CoffeeScript: it's not the implementation but the syntactical sugar that's driving its explosive growth. Consider this – modern browsers support most features provided by jQuery, but jQuery hasn't missed a beat in terms of widespread use. CoffeeScript is in the same boat. It just makes JavaScript less painful, even fun, and TypeScript doesn't.

FACT FILE

TRY IT ONLINE

Both libraries' sites offer in-browser compilers: enter CoffeeScript or TypeScript and see the corresponding JS.

INSTALLATION

TypeScript
npm install -g typescript

CoffeeScript
npm install -g coffeescript

CONTRIBUTE

CoffeeScript: *github.com/jashkenas/coffee-script*

TypeScript: *github.com/Microsoft/TypeScript*

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/howarth-257*

ABOUT THE AUTHOR

BENJAMIN HOWARTH

w:*about.me/benjaminhowarth*

t: @benjaminhowarth

areas of expertise:
a: Open source frameworks on the Microsoft .NET stack

q: What makes you squeamish?
a: Moths. Especially when they fly round your head – fuzzy things near my ears make me want to run away very fast

SPA

TRANSFORM YOUR WEBSITE INTO AN SPA

Benjamin Howarth explains how to take an accessible website and turn it into a scalable, manageable single-page app

As the plethora of devices, platforms and browsers grows larger every day, single-page applications (SPAs) are becoming more and more prevalent. HTML5 and JavaScript are now first-class citizens in application development, as well as revolutionising how we design and create great website experiences with CSS3.

As a result, the traditional pattern for developing a comprehensive 'digital strategy' is changing fast, and becoming a lot more streamlined. Platforms like PhoneGap enable us to create apps in HTML5 and JavaScript, which can then be adapted and cross-compiled for various mobile platforms (iOS, Android, Blackberry and Windows Phone). However, gaps still persist between languages and frameworks – especially between building 'apps' for mobile and tablet platforms, and building traditional, accessible and progressive websites.

As a freelancer, I've experienced clients asking for a website, then a mobile website, then an app (I built Kingsmill Bread's mobile and desktop/tablet websites on the MIT-licensed, .NET-based Umbraco CMS). The 'traditional' route of building this would be to do the site first, with a responsive (or specially-designed) CSS layout, then build a separate app that plugs into an underlying content store.

This model works great if you have specialists in iOS, Java and Silverlight (yes, Silverlight – the

VIDEO

See this SPA tutorial in action in Benjamin Howarth's exclusive screencast at: *netm.ag/tut1-257*

Sliced bread The Kingsmill Bread desktop and mobile websites share content using the Umbraco CMS, with user-agent specific templates

joys), but falls flat when it comes to scalability and manageability. Factor in the average cost of bespoke app development (approximately £22,000) and average return (£400 a year, if you're lucky – see *netm.ag/economics-257*), and suddenly you're pouring money into an awfully large technological black hole.

WHAT ABOUT JAVASCRIPT?

"JavaScript saves the day!" I hear the frontend developers cry. Not so fast. JavaScript solves the majority of the cross-platform issues, but then you're not building a website any more. Google doesn't index content loaded with AJAX, so you've lost accessibility and progressive enhancement: the key tenets of good web architecture.

The traditional pattern for developing a digital strategy is becoming a lot more streamlined

What's that I hear? "NodeJS saves the day! JavaScript everywhere!" Hold your horses there, hipster. NodeJS doesn't yet have a JavaScript-based CMS. "What about all those lovely, stable, mature content management systems that make websites awesome for their content managers?" Are you seriously suggesting direct-porting WordPress, Drupal, Umbraco et al to JavaScript just to solve this problem? (I had someone suggest this solution to me at the Scotch on the Rocks conference this year, and my reply was pretty much exactly that).

So how do we design a website that's accessible, and then 'upgrade' it to look, feel and behave like

▶

IN-DEPTH

WHAT'S A PROMISE?

In functional programming languages, a promise is effectively a proxy for lazy loading, thus preventing any operations from blocking your execution (and subsequent debugging). A perfect example from their documentation is replacing this:

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // Do something with value4
      });
    });
  });
});
```

With this:

```
Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
  // Do something with value4
})
.catch(function (error) {
  // Handle any error from all above steps
})
.done();
```

This enables you to manage your functions asynchronously, allowing each to run in its own thread, and making debugging more manageable. We can also manage UI updates while doing background work. BreezeJS uses Q to allow us to write this:

```
var client = new BreezeManager("baseURI")
.query("RestfulURIToQueryMyData")
.success(function(data) {
  // do something with data
}).fail(function(data) {
  // throw an error
}).complete();
```

It gives us separation in an inversion-of-control process in a functional language. This is useful for complex UIs, that have rich interaction with backend JSON services. Here we're using it to show you how to interact with database data on the server, served up by Microsoft's Web API stack.

an SPA? I'm glad you asked that, because that's what we're going to build in this tutorial.

Architecture is everything, I will cover a simple blog site, which you can then go on to adjust according to your project. I'm a .NET bod by trade, and it's worth noting this tutorial is partially written in .NET for the server-side code. If you don't like that, the design patterns will easily translate into an MVC framework of your choice – Zend, CodeIgniter, Ruby, Django and so on.

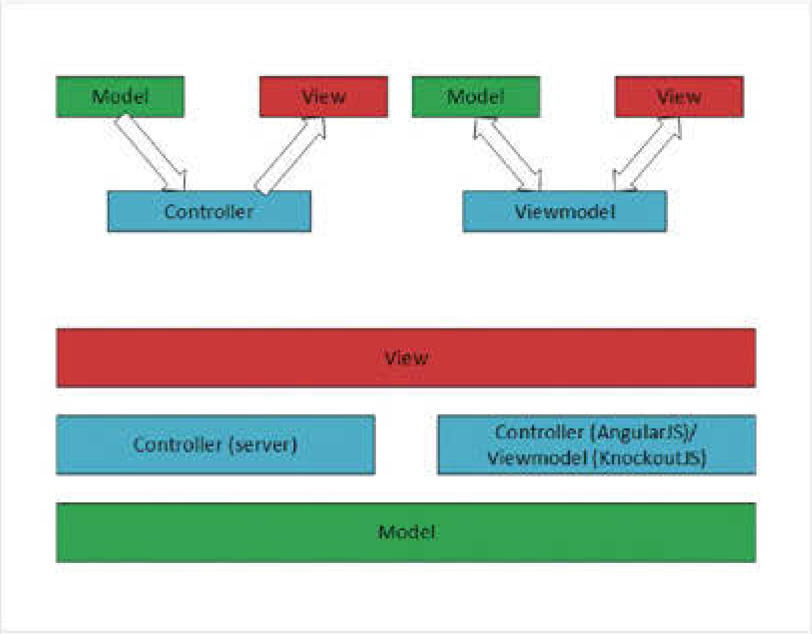
BRIDGING THE GAP

Most websites nowadays are built with a model-view-controller (MVC) architecture, whereas apps, which rely on data loaded on-demand, are built with a model-view-viewmodel (MVVM) architecture. The two paradigms are related, but differ slightly in event handling. Whereas a controller would have an action to handle a particular event (submission of data), a viewmodel would have a specific event defined for handling UI updates and any model binding modifications (people familiar with C# will know this as the INotifyPropertyChanged pattern).

The key tenet in this architecture is working out how to share the model and the view between the two patterns. When designing your site in this fashion, 99.9 per cent of the time your views will be identical both on the server and client sides. The only difference is when and how they are populated with model data – be it by controller on the server, or controller on the client.

We'll start off with creating a simple website (which can be downloaded at *netm.ag/demo-257*) using ASP.NET MVC 4. It's got a couple of controllers in it, a model created with Entity Framework to show an example of server-side model population, and some views with a responsive CSS3 template built with Bootstrap. It contains everything on the left-hand side of the diagram below – a basic model, a controller and a couple of views, just to get everything started.

Both sides Architecture of MVC and MVVM design patterns, and how we're going to try and bring them together across the client and the server



IT'S A BREEZE

We're now going to install AngularJS and BreezeJS into the project. NuGet (the package manager for Visual Studio) makes this simple to add into our project. BreezeJS depends on Q, a tool for making and managing asynchronous promises in JavaScript.

To create a very simple SPA experience from our existing site foundation, we need to share the routing table in MVC on the server with AngularJS.

Sites are built with MVC architecture, whereas apps are built with MVVM architecture

We'll do that using a controller that serialises and spits out the routing table in a format that Angular will understand.

Angular expects routing data to be configured as follows:

```
when('/blog', {
    templateUrl: 'partials/blog.html',
    controller: 'BlogListCtrl'
  })
.when('/blog/post/:blogId', {
    templateUrl: 'partials/blog-post.html',
    controller: 'BlogPostCtrl
  })
.otherwise({
    redirectTo: '/index'
  });
```

As we can see from this code, Angular's routing provider expects to know both the controller and the corresponding view. We'll be feeding AngularJS a generic controller to catch most of our requirements, and then extending it for particular views (i.e. blog list and blog post, as these contain more than just vanilla data).

To do this, we need to identify which controllers (and corresponding routes) are to be fed from the server into Angular. This is where we take advantage of .NET's reflection capabilities – code that reads code, or ‘code-ception’ as I like to think of it. Warning: here be dragons! We’ll be marking our controllers on the server. This is to be shared with the JS side of the site, with metadata, using a .NET feature called Reflection to gather these controllers up. On these controllers, we’ll be marking out our actions with metadata to identify the corresponding JS view, model and controller, as shown in the diagram on the right.

This approach is not foolproof. If you have filters or constraints on the routes or controller actions, whatever is provided by the JS routeProvider as Angular routes may not actually end up being rendered by the server. For more information on advanced MVC routing, route constraints and filtering, I highly recommend checking out Scott Allen’s courses on Pluralsight on the ASP.NET MVC stack (*netm.ag/allen-257*).

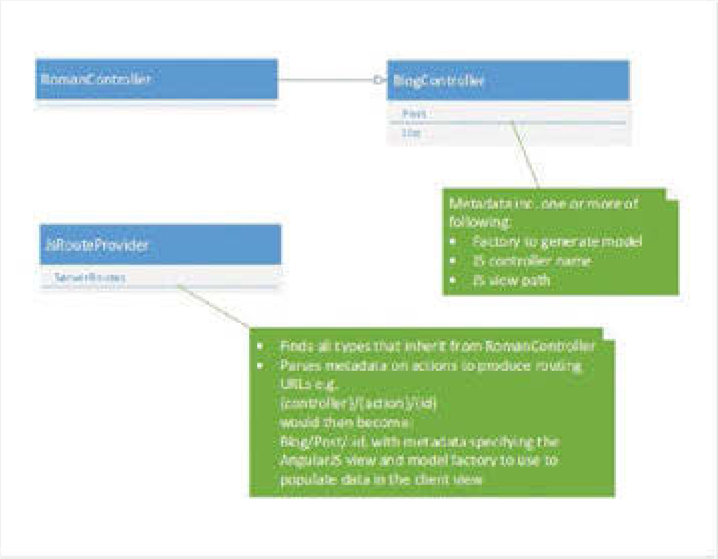
PIECE BY PIECE

Now we’ve provided Angular with our routing table, we need to start gluing it together. Firstly, we update our controllers to inherit from our generic RomanController. We’ll be overriding some parts of this later, but for now it’s a generic marker for us to identify the routes and URLs we want Angular to have access to.

Next, we need to mark the actions we want to be available in AngularJS, with a custom attribute: `RomanActionAttribute` .

```
public class RomanDemoController : RomanController {
    private RomanSPA.Demo.Models.
RomanSPAStarterKitEntities _context;
    public RomanDemoController() : base() {
      // Yes, I'm not using dependency injection for my DB
context, cause this is a demo ;-)
      if (_context == null) _context = new Models.
RomanSPAStarterKitEntities();
    }
    [RomanAction]
    public ActionResult Index() { return View(new
IndexModel()); }
    [RomanAction(Factory=typeof(BlogListFactory), Controlle
rName="BlogController", ViewPath="/assets/blog-list.html")]
    public ActionResult Blog() { return View(_context.
BlogPosts); }
    [RomanAction(ControllerName="BlogPostController")]
    public ActionResult BlogPost(string slug) {
      if (_context.BlogPosts.Any(p => MakeTitleUrlFriendly(p.
Title) == slug)) {
        return View(_context.BlogPosts.First(p =>
MakeTitleUrlFriendly(p.Title) == slug));
      } else {
        return HttpNotFound();
```

▶


RomanController
BlogController
Post
List
JsRouteProvider
ServerRoutes
Metadata inc. one or more of following:
▪ Factory to generate model
▪ JS controller name
▪ JS view path
▪ Finds all types that inherit from RomanController
▪ Parses metadata on actions to produce routing URLs e.g.
{controller}/{action}/{id}
would then become:
Blog/Post/:id, with metadata specifying the AngularJS view and model factory to use to populate data in the client view

Populating routes A UML diagram of how our routes are populated from the metadata on each action, which we specify with RomanActionAttribute

★ FOCUS ON

ACTION FILTER?

Microsoft’s ASP.NET MVC platform is a framework that demonstrates an aspect-oriented programming (AOP) pattern. AOP frameworks allow you to decorate your code with metadata attributes that then point to methods which fundamentally alter the behaviour of the code being executed at runtime.

Examples of AOP frameworks include Spring for Java (and Spring.NET for us MS bods, which is a direct port), FLOW3 and PECL support for AOP in PHP, and a commercial tool I use called PostSharp that’s a .NET ‘inter-weaver’ for adding AOP to non-MVC projects, by wrapping around code as you’re compiling it.

An action filter is an example of a cross-cutting concern in AOP. In the ASP.NET MVC stack, it can be used to modify the result of an action on a controller. In this tutorial, I use it to potentially serve up different data, depending on the parameters supplied.

ASP In the ASP.NET stack, action filters can change the result of an action

```
    }
  }
  [RomanAction]
  public ActionResult About() { return View(); }
  private string MakeTitleUrlFriendly(string title) {
    return title.ToLower().Replace(" ", "-");
  }
}
```

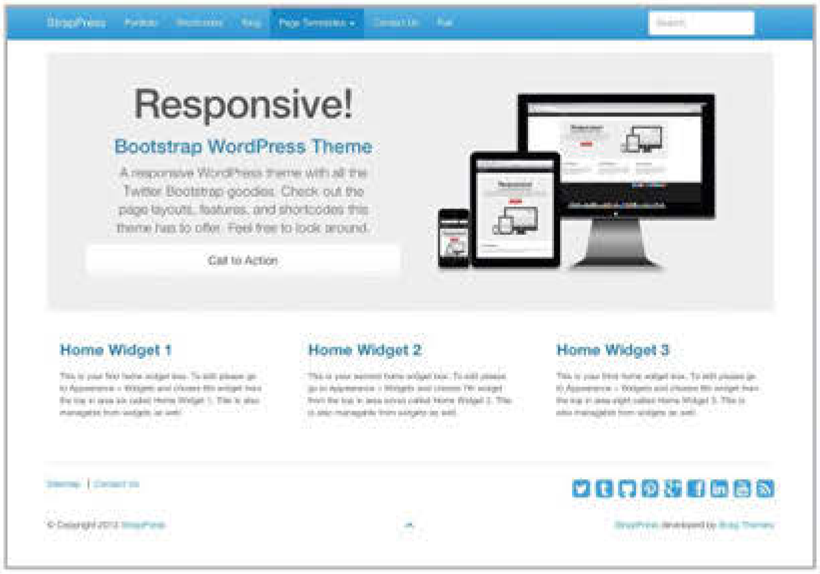
This attribute is an action filter, and has three parameters, all optional: model factory, controller name and view name. This gives our routing table the option to specify any explicit overrides we want, or to go with a generic, default behaviour (i.e. load a controller with no JSON model data, which then loads a partial view from the server). Note how `[RomanActionAttribute]` can be automatically shortened to `[RomanAction]` by the C# compiler.

The fundamental part of this exercise is that we should be able to share views and expose server data for complex transformations in our app. This means we can take advantage of HTML5 offline caching to create a great user experience on the app side.

We'll leave the `Index` and `About` views blank. For the `BlogList` action, we're going to specify a factory, a custom controller and a custom view location for the client-side view. For the `BlogPost` action, we'll specify a custom controller to load the individual post from the JSON we get from our BlogList factory.

Now, we set up a `GenericController` in AngularJS, in `/App/` controllers, and we set up auto-routing in AppJS so that all the routes are provided from the server to the client. You can provide extra routes manually by overloading the `RoutesApiController` and overriding the `ExtraRoutes` property with your own custom list of routes to pass to AngularJS.

Templates Bootstrap is the go-to CSS framework for a responsive site these days, spawning dozens of online template markets



Responsive!

Bootstrap WordPress Theme

A responsive WordPress theme with all the Twitter Bootstrap goodies. Check out the page layouts, features, and shortcodes this theme has to offer. Feel free to look around.

Call to Action

Home Widget 1

Home Widget 2

Home Widget 3

```
angular.module('RomanSPA', ['ngRoute'])
    .config(['$routeProvider', function ($routeProvider,
$locationProvider) {
      $.ajax({
        url: '/api/RouteApi/AllRoutes',
        success: function (data) {
          for (var i = 0; i < data.length; i++) {
            $routeProvider.when(data[i].RoutePattern, {
              controller: data[i].controller,
              templateUrl: data[i].templateUrl
            });
          }
          $routeProvider.otherwise({ redirectTo: '/' });
        },
        fail: function (data) {
        }
      });
    }])
    .value('breeze', window.breeze);
```

CORE FUNCTIONS

Finally, in our GenericController, we put in two core functions – one to manually apply our specified template, and one to retrieve the model using the factory on the action attribute.

```
angular.module('RomanSPA')
    .controller('GenericController', ['$scope', '$route', function
($scope, $route){
      $scope.pageUrl = '/';
      if ($route.current !== undefined) { $scope.pageUrl =
$route.current.templateUrl; }
      // Retrieve our view - be it from server side, or custom
template location
      $.get({
        url: $scope.pageUrl.toString(),
        beforeSend: function(xhr) { xhr.setRequestHeader('X-
RomanViewRequest', 'true'); },
        success: applyView
      });
      // Retrieve our model using the modelfactory for our
current URL path
      $.get({
        url: $scope.pageUrl.toString(),
        beforeSend: function (xhr) { xhr.
setRequestHeader('X-RomanModelRequest', 'true'); },
        success: applyModel
      });
      function applyView(data) {
        $scope.$apply(function () { angular.element('body').
html($compile(data)($scope)); });
      }
      function applyModel(data) { $scope.model = data; }
}
    }]);
```

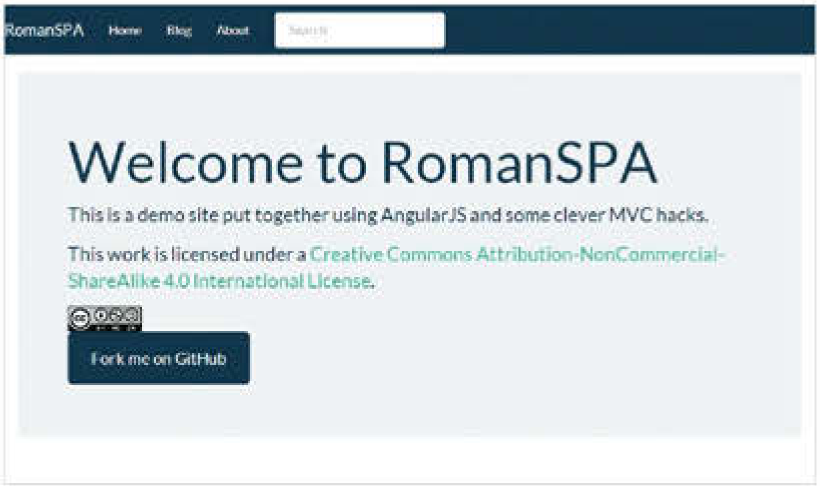
Finally, we're at a point where:

- We have gathered up all the routes on the server that we want AngularJS to take advantage of. These will be exported to Angular's routing library when our app boots up.
- We have specified a generic controller, from which our specific controllers can inherit. However, this means we have 'generic' SPA functionality, with pages partially loading as we navigate through the site.
- Child actions that we may want AngularJS to have access to, but shouldn't be in the routing table (such as navigation, footer and other partial views), can be marked out with `[RomanPartial]`.
- A request to `/RomanDemo/Index` will give us a full page, but when AngularJS requests it, it'll either provide a partial view or a JSON object, depending on the metadata we have supplied.
- Actions we want to specify metadata for (or export to Angular routing) – i.e. custom JSON model, custom template URL or custom AngularJS controller – are marked with `[RomanAction]`.

> We can take advantage of HTML5 offline caching to create a great app user experience

We're almost set. From here, to guide you in the kind of direction you can go with your hybrid site app, we'll create a BlogController that inherits from `GenericController`, and use this to gather up all the blog posts we want. We'll then use this as a way to enable users to navigate a blog, whilst most of the data is held in HTML5 offline storage for us.

```
angular.module('RomanSPA')
   .controller('BlogController', ['$scope', function ($scope) {
      $controller('GenericController');
      function storePostsOffline(data) {
         if (!supportsLocalStorage()) { return false; }
         localStorage["RomanSPA.data.blog.posts"] = data;
         return true;
      }
      var manager = new breeze.EntityManager('/breeze/
BlogApi');
      var query = new breeze.EntityQuery('BlogPost');
      manager.executeQuery(query)
         then(function(data) {
            storePostsOffline(data);
            $scope.posts = data;
         })
         .fail(function(data) {
            // silently fail for now
         });
   }]);
```


RomanSPA Home Blog About Search

Welcome to RomanSPA

This is a demo site put together using AngularJS and some clever MVC hacks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Fork me on GitHub

In action The initial site template booted up, with a quick Bootstrap theme from *bootswatch.com*

HTML5 offline storage is the biggest reason you'd want to do this. You can also extend the view engine to automatically store templates and views offline. By doing this, you're allowing your website visitor to treat your website more like an app – which they can use even while they're without WiFi or phone signal.

WRAPPING UP

We now have:

- A basic MVC site, with routes for basic pages and a blog.
- An MVC app that sits on top, powered by AngularJS, with a matching route table.
- AngularJS making AJAX requests using jQuery, which server-side MVC then interprets to return JSON model or HTML view appropriately.

The result of this mashup? The ASP.NET MVC RomanSPA framework! (vomitorium, sickle-scrapers and sulphur-tasting water all optional.) You can use this foundation to build out many more complex parts to your site.

The core framework, and extensions for both AngularJS and KnockoutJS or BreezeJS, have been 'forked' into separate projects. You can either install the AngularJS or KnockoutJS editions, which will get you kick-started straight away with all the necessary prerequisite libraries installed, or you can simply install the core and write your own extensions for EmberJS or any other MVx JavaScript framework that takes your fancy. ◼

RESOURCE

NUGET

NuGet deals with dependency management and downloading to keep your development simple. The packages for this tutorial can be found here: *netm.ag/NuGetAngular-257* *netm.ag/NugetKnockout-257*

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/spa-263*

ABOUT THE AUTHOR

JOSH EMERSON

w: *joshemerson.co.uk*

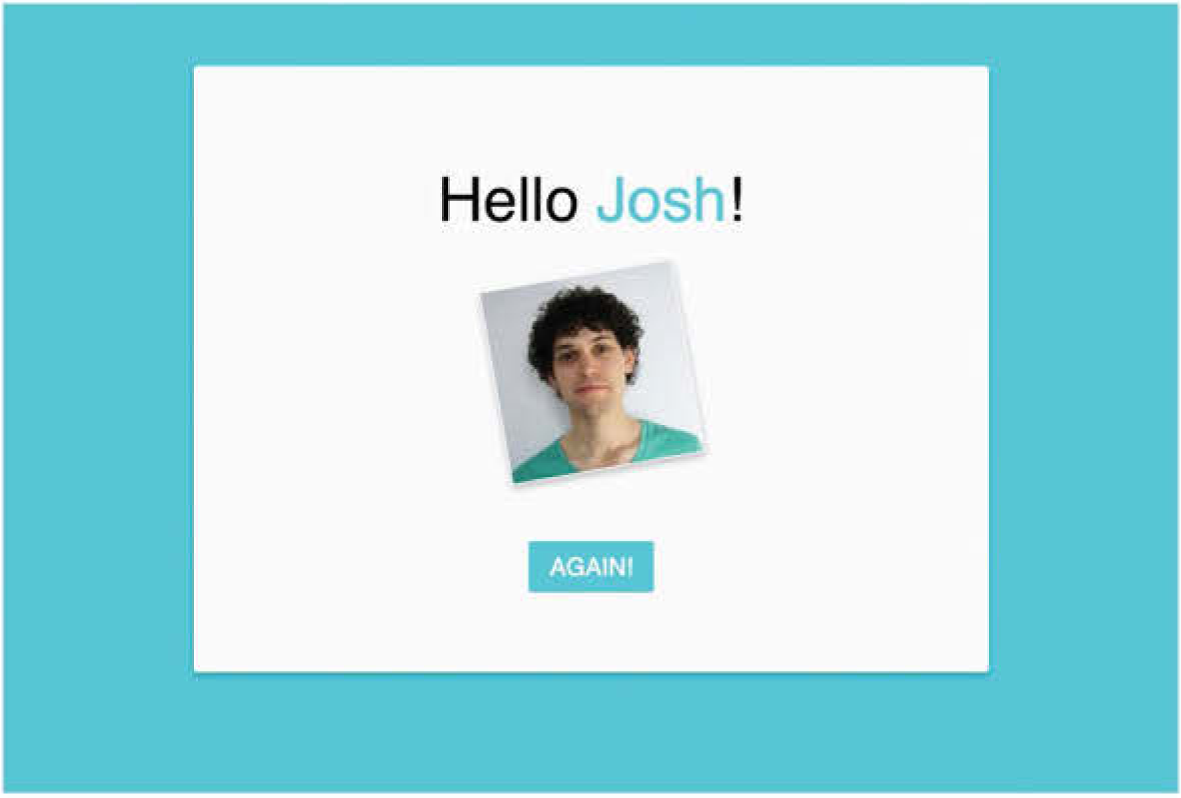
t: @joshje

job: Frontend developer, Mendeley

areas of expertise: HTML, CSS, JavaScript, RWD, progressive enhancement

q: what's the dumbest thing you've done this year?

a: I forgot my passport on a trip to Belfast. After a major panic at the airport, I remembered Northern Ireland is part of the UK, and I didn't need it at all!



Hello Josh!

✱ SPA

ENHANCE YOUR SINGLE PAGE APPLICATION

Josh Emerson explains how to create a site that works with or without JS, using the same templates on the server and client

VIDEO

Josh Emerson has put together an exclusive screencast to go alongside this tutorial. Watch along at *netm.ag/spavid-263*

The web has come a long way since the static documents it was originally associated with. When we think of a 'modern' experience on the web, we often think of websites that are responsive and adapt as we use them. These sites are often referred to as 'web applications' because they have so much application logic and are very different when compared to static, document-centric sites.

Increasingly, the logic that powers these websites is all taking place on the frontend, using JavaScript. Once you start building your website using JavaScript, there's no need to load any HTML beyond the initial page load. All subsequent requests can be made via AJAX and by simply modifying the DOM. This type of website is known as a single-page application, or SPA.

This idea of doing everything in JavaScript works well when users have powerful machines and are using recent browsers that have consistent APIs. However, even amongst recent browsers there are major differences in API implementations, and many people are now browsing the web on mobile devices with very slow processors. That's not to mention those users on a portable device with an intermittent connection that may not even manage to download and execute the JavaScript at all.

With SPAs, we need to ensure the URL is modified as a user navigates the website, so that they can do things like link to a page, and use their browser's back and forward buttons. Ensuring a website loads the correct page from a URL is easy when the page is served as HTML, but if your payload is an empty HTML document (as is the case with most SPAs), your JavaScript needs to put the website back into the correct state. This will likely require additional API requests, and will slow down the loading of the page, not to mention increasing its complexity.

JAVASCRIPT ENHANCEMENT

However, there is another way. We can create a website that loads plain HTML pages which are then enhanced via JavaScript. All subsequent navigations are enhanced using JavaScript, updating the URL using the History API. Airbnb calls this technique 'Isomorphic JavaScript' and wrote a blog post about how it could be used to render the same views on the server and the client (*netm.ag/isomorphic-263*).

In this tutorial, I will show you that it is possible to have your JavaScript-flavoured cake and eat it too. We will create a Node Express server that uses exactly the same templates as the client to render pages. The templates used in this tutorial are written in Mustache (*mustache.github.io*) and can be rendered by a number of server-side languages, including Ruby, Python, Node, PHP and Perl.

We can create a site that loads plain HTML pages which are then enhanced via JS

I chose Node because it means we can write JavaScript on the server and client. Rather than creating our own API, we will be using the Gravatar API (*en.gravatar.com*). This requires an email address and returns a JSON file containing the user's profile image, if they have an account. I've created an example at *shared-templates.joshemerson.co.uk*.

CREATING THE NODE SERVER

First off you'll need to ensure you've got Node installed on your machine. If you haven't, go to *nodejs.org/download* and choose the installer for your system. Now you can get started.

In the GitHub repository at *netm.ag/template-263* you can see a file called `package.json`. Copy this file to a folder where you want your project to live. It describes the project and its dependencies. In a terminal window, `cd` into your project folder and type `npm install` to install the dependencies. Once

▶

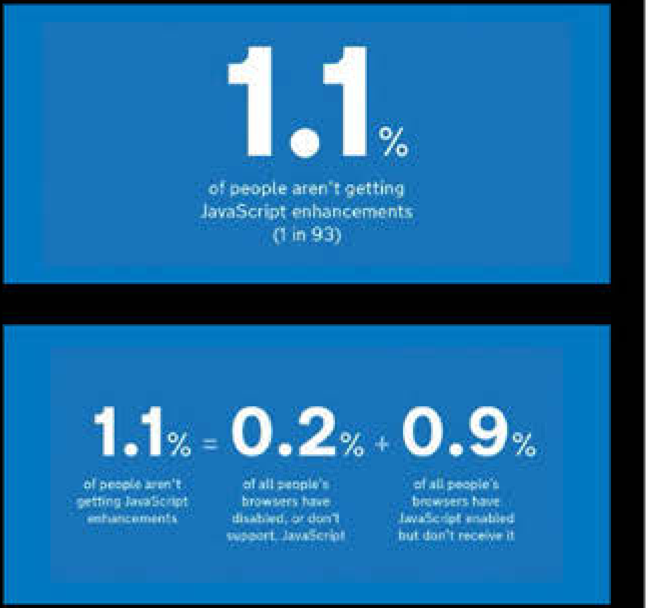
FOCUS ON

DOESN'T EVERYONE HAVE JAVASCRIPT NOW?

... well, not according to the UK Government. A Government Digital Services (GDS) blog post (*netm.ag/JS-263*) reported that 1.1 per cent of its users (made up of the UK population at large) didn't receive JavaScript. This was made up of 0.2 per cent of people whose browsers didn't support JavaScript or had explicitly disabled it, and a massive 0.9 per cent of users who had JavaScript enabled, but for some reason did not receive it successfully (perhaps their train went into a tunnel – it's happened to us all). So, with any website, it's good to offer some baseline experience for users without JavaScript.

Another good reason to ensure your site works without JavaScript is that an empty page doesn't get a great deal of Google juice. It's much better to ensure that the same content is available in the no-JavaScript version. OK, so apparently Google's spiders are smart enough to run JavaScript these days, but what about other web-crawling bots?

One last issue with rendering a page in the browser is the performance hit on first load. Instead of downloading some HTML and applying CSS to it, the browser has to also execute some JavaScript and then assemble the page. This critical path to first load is further delayed as the browser usually begins downloading any inline images before render. With JavaScript, it has to wait until after the page content has been appended to the DOM. You can read more about this at *netm.ag/performance-263.*



Stats Research from the GDS revealed some interesting facts about JavaScript that's done, you'll notice a node_modules directory containing those dependencies.

Let's create our app.js file in our project folder. Type the following into your text editor of choice:

```
var express = require('express');
var hogan = require('hogan-express');
var request = require('request');
var md5 = require('MD5');
```

This will require our dependencies and make them available to the app. If you want to learn more about these dependencies, go to *npmjs.org* and search for the package you're interested in.

Let's configure our server. In app.js, write:

```
var app = express(); // This is our express app
app.set('view engine', 'mustache'); // Templates have an extension of *.mustache...
app.set('views', 'templates'); // and reside within the / templates directory
app.set('layout', 'layout'); // Use templates/layout.mustache for the page layout
app.engine('mustache', hogan); // Use hogan to render mustache templates
```

Copy the templates directory in the GitHub repo to your project folder, and take a look at the files.

- layout.mustache will be used as the base layout for every page and serves as the starting point for the document. The contents of the individual page will go where it says {{{yield}}}
- The template form.mustache is used for showing a form that asks the user for their email address
- Once the user has submitted the form we will use the template hello.mustache to show the user's name and photo
- Finally, loading.mustache will be used later in our JavaScript enhancements in order to show that some network activity is occurring

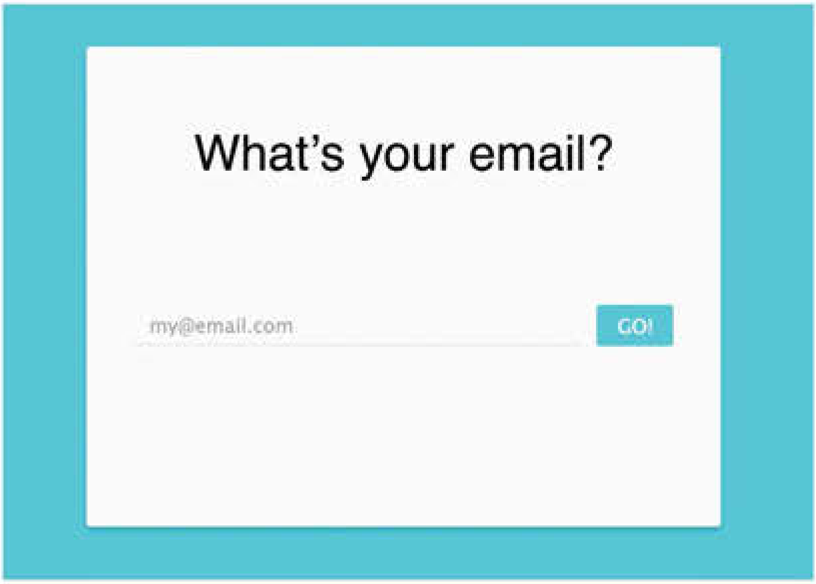
Make a public directory in your project folder and copy across the public/css directory that contains our CSS styles.

EMAIL TESTS

Now it's time to make our server actually do something. Inside app.js, write the following:

```
app.get('/', function(req, res) {
 // Render a form for all requests to the root (/)
  res.render('form');
});

// Make the /templates folder available to the web at / templates
app.use('/templates', express.static(__dirname + '/ templates'));
// Make bower components available at /bower_components
app.use('/bower_components', express.static(__dirname + '/ bower_components'));
// Make the /public folder available to the web at /
app.use(express.static(__dirname + '/public'));

app.get('/hello', function(req, res) {
 // Get the email query parameter
  var email = req.query.email;
  // MD5 hash it
  var hash = md5(email.replace(/^\s+|\s+$/g, ''));

 // Fetch the user info from Gravatar API
  request({
    url: 'http://www.gravatar.com/' + hash + '.json',
    headers: { 'User-Agent': 'Node' }
  }, function (err, response, body) {
    if (err || response.statusCode !== 200) {
     // On error, render a form with an error message
      res.status(response.statusCode);
      res.render('form', {
        error: true,
        email: email
      });
    } else {
     // Render the hello template with the user data
      var json = JSON.parse(body);
      var user = json.entry[0];
      res.render('hello', user);
    }
  });
```

Email form The 'form' template renders a form where the user can enter and submit their email address


What's your email?

```
});

// Start our server on port 3000
var server = app.listen(3000, function() {
  console.log('Listening on port %d', server.address().port);
});
```

You can now run the server by typing `node app.js` into your console. Go to *http://localhost:3000* in your browser, and all being well you should see your form. Try submitting the form with your email address. Notice that it sends a `GET` request to the `/hello` endpoint. Our server uses the email address you have provided and attempts to fetch the user info from the Gravatar API.

Depending on whether the email is found by Gravatar or not, at this point you should either see the form (with an error message), or the user's photo. If you don't have a Gravatar account, you can try using this test email address instead: example@joshemerson.co.uk.

Our server uses your email address and attempts to fetch the user info from Gravatar

THE CLIENT SIDE

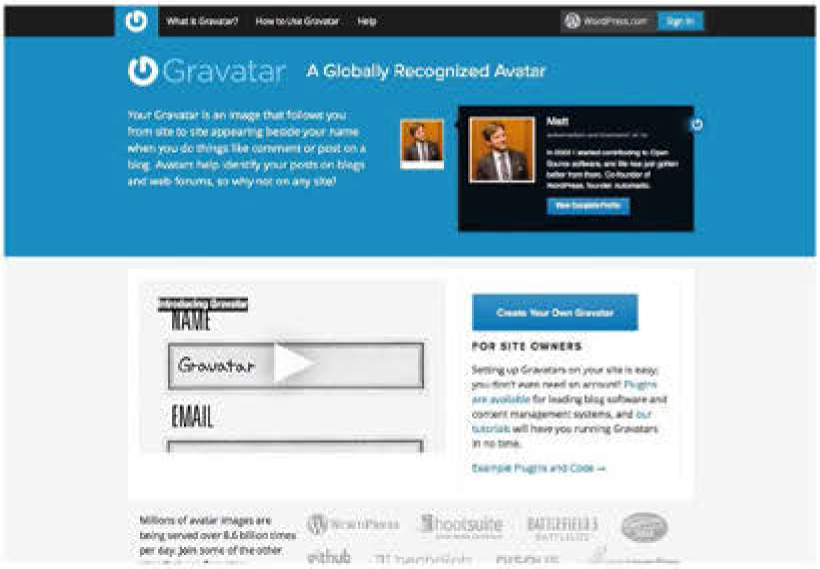
That's it for the server side, but what about the client-side app experience? First, let's get our client-side dependencies using Bower. Copy the file `bower.json` from the GitHub repository to your project folder. This will fetch client-side dependencies, including RequireJS, which will allow us to write our code in an AMD style. You can find out more about RequireJS at *requirejs.org*.

Next we need to write our client-side JavaScript. But before we do that, let's install our client side dependencies. These are similar to the npm modules we installed earlier, but they work in the browser.

If you haven't used Bower before, install it globally by typing `npm install bower -g`. Then install the bower dependencies by running `bower install`.

You'll notice a folder called `bower_components` that contains libraries used by our client-side JavaScript. Now we will create the client side JavaScript. We will end up with three files:

- `gravatar.js` will handle API calls to Gravatar and will cache the response
- `routes.js` will handle rendering routes in the client
- `main.js` will contain all the business logic



Your Gravatar is an image that follows you from site to site appearing beside your name when you do things like comment or post on a blog. Avatars help identify your posts on blogs and web forums, so why not on any site?

FOR SITE OWNERS

Setting up Gravatars on your site is easy; you don't even need an account! Plugins are available for leading blog software and content management systems, and our tutorials will have you running Gravatars in no time.

Example Plugins and Code →

Millions of avatar images are being served over 8.6 billion times per day. Join some of the other

Gravatar The Gravatar API (*en.gravatar.com*) requires an email address and returns a JSON file containing the user's profile image

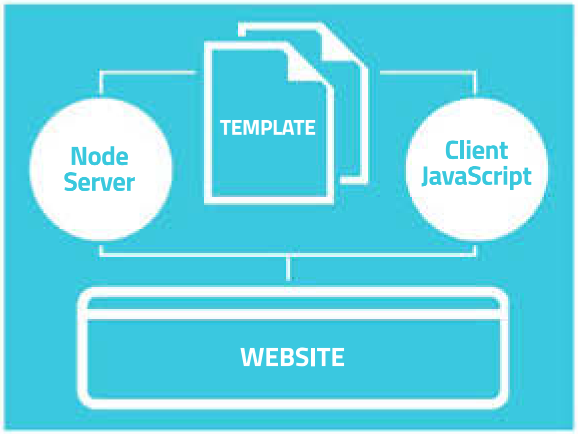
All of these files need to reside in the `public/js` directory. For the purposes of this tutorial, I won't go into the `gravatar.js` file. Simply copy it from the GitHub repo to `public/js`.

`public/js/main.js` should look as follows:

```
// Configure paths to our dependencies:
require.config({
  baseUrl: '/js',
    paths : {
    text: '/bower_components/requirejs-hogan-plugin/text',
    hogan: '/bower_components/requirejs-hogan-plugin/hogan',
    hgn: '/bower_components/requirejs-hogan-plugin/hgn',
    md5: '/bower_components/md5-jkmyers/md5.min'
  }
});

// Require the modules we need here:
require([
  'gravatar',
  'routes'
], function(
  gravatar,
  routes
) {
  var content = document.querySelector('.content'); // We will bind events to this element

  // Get a user from Gravatar and then render them
  var renderUser = function(email) {
   // Render the loading screen
   routes.loading();

   gravatar.getUser(email, function(error, user) {
    if (error) {
```

▶

```
   routes.render('/', {
     error: true,
     email: email
   });
  } else {
   routes.render('/hello', user);
  }
 });
};

// This event fires as a user navigates the history:
window.addEventListener('popstate', function(e) {
  var state = e.state || {};
  if (state.getUser) {
    renderUser(state.email);
  } else {
    routes.render(window.location.pathname, state);
  }
});

// Handle link clicks and use the history API
content.addEventListener('click', function(e) {
  var el = e.target;
  var url = el && el.getAttribute('href');
  if (routes.has(url)) {
    e.preventDefault();
    window.history.pushState({}, '', url);
    routes.render(url);
  }
}, true);

// Handle the form submission
content.addEventListener('submit', function(e) {
  var emailEl = e.target.querySelector('input.email');
  if (emailEl) {
    var email = emailEl.value;
    e.preventDefault();
    // Store the email in this current history state
    window.history.replaceState({
```



Templates The same templates are used to generate the first page on the server, and for navigation to happen in the browser

IN-DEPTH

CUTTING THE MUSTARD

Given that our website works both with and without JavaScript, we are now in a position to offer the non-JavaScript version to more users. That might seem a bit backwards. Why would you remove JavaScript from a browser that supports it? Well not all browsers are created equal, and often supporting a browser comes at a price. All browsers will work with the non-JS version, but things like History API, localStorage and various other JavaScript APIs may not be available, or may require polyfills.

This technique for switching off JavaScript enhancements based on feature detection is known as 'Cutting the Mustard' (*netm.ag/mustard-263*). Using this technique will speed up development and make your code easier to maintain, as there are only two clearly defined experiences, rather than a myriad of fallbacks and browser-hacks to patch support for incompatible browsers.

Even more importantly, you won't be including unneccesary code and polyfills for browsers that do support the APIs you require. Being future-friendly (*futurefriendlyweb.com*) is not just about recognising the uncertainty of the future, it's also about not being held back by the past.

If you look at the bottom of `templates/layout.mustache`, you'll notice that we checked for three browser APIs before requiring our main JS file:

- `document.querySelector`
- `window.localStorage`
- `history.pushState`
- `window.addEventListener`

Our app depends on these APIs. But because the whole website works without JavaScript, we can just not load the JavaScript on incompatible browsers. These users will get the simple version of the website.

FUTURE FRIENDLY

In today's incredibly exciting yet overwhelming world of connected digital devices, these are the truths we hold to be self-evident:

Disruption will only accelerate. The quantity and diversity of connected devices—many of which we haven't imagined yet—will explode, as will the quantity and diversity of the people around the world who use them. **Our existing standards, workflows, and infrastructure won't hold up.** Today's onslaught of devices is already pushing them to the

Future friendly Web pros have written up a future-friendly philosophy

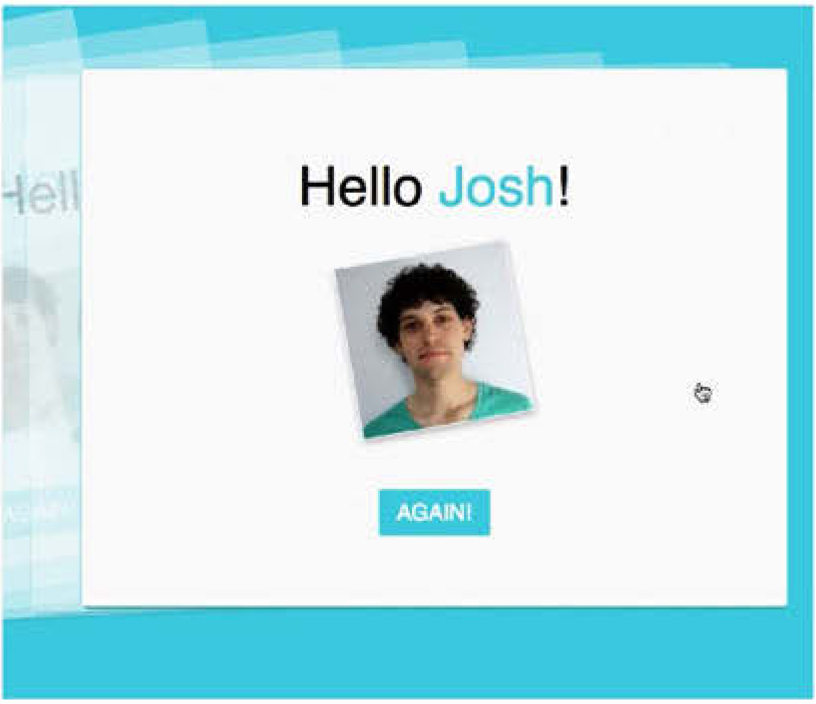
```
        email: email
      }, '', '/');
      // Update the URL to the /hello page
      window.history.pushState({
        getUser: true,
        email: email
      }, '', '/hello?email=' + encodeURIComponent(email));
      // Get the new user
      renderUser(email);
    }
  }, true);
});
```

This file does a lot. It provides a function to fetch a user using the Gravatar API. It also binds to several events in order to update the view and fetch the user at the correct moments.

Now, let's take a look at the `public/js/routes.js` file. This file was called several times in `main.js` and renders routes.

Try disabling your JavaScript and refreshing ... the page should look the same

```
// Require the templates we need
define([
  'hgn!../templates/form',
  'hgn!../templates/hello',
  'hgn!../templates/loading'
],function(
  form,
  hello,
  loading
) {
 // Map routes to templates
  var routes = {
    '/': form,
    '/hello': hello
  };
  // Find the elements we need to render templates
  var wrap = document.querySelector('.wrap');
  var content = wrap.querySelector('.content');

  return {
   // routes.render renders a path with optional data
    render: function(path, data) {
      data = data || {};
      // Animations!
      wrap.className += ' animate-out';
      setTimeout(function() {
        // We wait 500ms for the animation to complete before removing the class
        content.innerHTML = routes[path](data);
        wrap.className = wrap.className.replace('animate-out', '');
      }, 500);
    },
    // routes.loading renders the loading template
    loading: function() {
      content.innerHTML = loading();
    },
    // routes.has checks that a route exists
    has: function(path) {
      return routes[path] ? true : false;
    },

  };
});
```



Smooth moves Once an email address has been entered, the box smoothly slides off-screen to be replaced by a profile image

Now let's try this out! Run `node app.js` again and go to *http://localhost:3000*. Try entering some emails. Notice the smooth animations – this is all in-browser now. Once you've entered a few emails, try using the browser back and forward buttons. It just works!

Notice how the URL bar updates because we're using the History API. Try disabling your JavaScript and refresh. No matter which state you are in, the page should look the same. So there we have it, a website that works with and without JavaScript, and shares the same templates on the server and in the browser.

Download the files here!

Download all the files you need for this tutorial at *netm.ag/firebasegit-265*

ABOUT THE AUTHOR

LUKAS RUEBBELKE

t: @simpulton

w: *onehungrymind.com*

Job: Senior developer advocate, Udacity

Areas of expertise: JavaScript, HTML, CSS, shenanigans

q: What's the best thing on your desk today?

a: An iced 'Lulu' – two shots espresso, two shots raspberry syrup and a can of Red Bull, on ice

ANGULARJS

BUILD A REAL-TIME APP WITH FIREBASE

Lukas Ruebbelke shows you how to take an AngularJS application and give it real-time capabilities using AngularFire

Imagine connecting an AngularJS application to a fully functional backend in a matter of minutes. Now imagine every client connected to your app would reflect changes to your data in real time. We are going to do exactly that, by taking an existing AngularJS leaderboard application, which only stores data in memory on the frontend, and converting it into a real-time web app using Firebase (*firebase.com*).

THE LEADERBOARD

First, let's look at how the static version of the leaderboard works. Our leaderboard consists of two main parts: the admin dashboard (where we manage contestant data) and the leaderboard (where we display contestant data in real time).

The leaderboard application consists of an AngularJS service called `ContestantsService` and a controller named `MainCtrl`. The `MainCtrl` delegates all operations that manipulate the `contestants` collection to the `ContestantsService`. This means when we add Firebase functionality to our application, all our changes will be contained in the service and nothing else needs to change.

Displaying contestants

We'll start with displaying contestants' data. In our `ContestantsService` we have created a `contestants` collection that we are making available via the `service.getContestants` method.

```
app.service('ContestantsService', function () {
  var service = this;
  var contestants = [
    {id: 1, lane: 1, name: 'Contestant 01', score: '10'},
    {id: 2, lane: 2, name: 'Contestant 02', score: '15'},
    {id: 3, lane: 3, name: 'Contestant 03', score: '20'}
  ];

  service.getContestants = function () {
    return contestants;
  };
});
```

Now we have access to the `contestants` collection, we will inject `ContestantsService` into the `MainCtrl` and assign the return value of `ContestantsService.getContestants` to `main.contestants`.

We are also initialising an empty `contestant` object to `main.newContestant`, which we will use to bind the new contestant form to.

```
// We are using the controllerAs syntax
app.controller('MainCtrl', function (ContestantsService) {
  var main = this;
  main.newContestant = {lane: '', name: '', score: ''};
  main.contestants = ContestantsService.getContestants();
});
```

With `contestants` available on our controller, we can use `ngRepeat` to display our `contestants` in a table.

```
<tr ng-repeat="contestant in main.contestants">
  <td>{{contestant.lane}}</td>
  <td>{{contestant.name}}</td>
  <td>{{contestant.score}}</td>
</tr>
```

VIDEO

Lukas Ruebbelke has created an exclusive screencast to go with this tutorial. Watch along at *netm.ag/firebasevid-265*

The leaderboard page is actually a subset of the admin dashboard, which allows us to use the same controller for both views.

Adding a contestant

In our dashboard, we will add a form that will allow us to add a new contestant. We are showing one input – to update the contestant's lane – but the format is the same for the contestant's name and score.

Here, we have a text input field that uses `ngModel` to bind to a property of `main.newContestant.lane`. We have also added `ngSubmit`, which calls `main.addContestant` when the form is submitted:

```
<form class="form-inline" role="form"
  novalidate ng-submit="main.addContestant()">
  <div class="form-group">
    <input type="text" class="form-control"
      ng-model="main.newContestant.lane"
placeholder="Enter lane">
  </div>
  <!-- The rest of the form -->
  <button type="submit" class="btn btn-default">Add</
button>
</form>
```

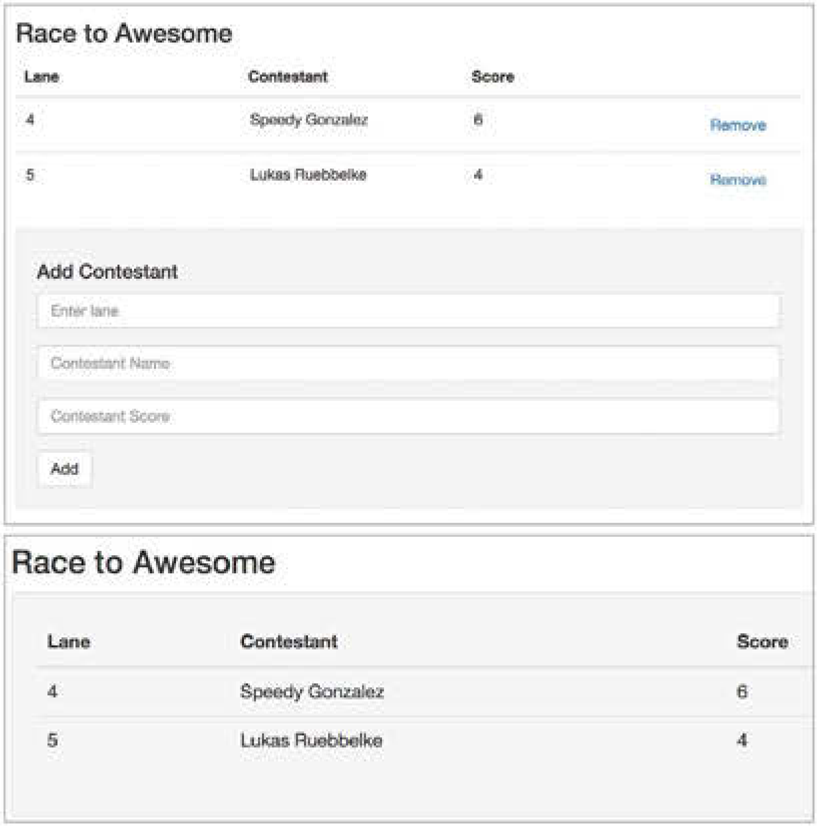
When `main.addContestant` is called, a copy of `main.newContestant` is passed on to `ContestantsService.addContestant` and `main.newContestant` is reset.

The leaderboard page is a subset of the admin dashboard, so we can use the same controller

```
main.addContestant = function () {
  ContestantsService.addContestant(angular.copy(main.
newContestant));
  main.newContestant = {lane: '', name: '', score: ''};
};
```

In the static version of the leaderboard, `service.addContestant` creates a fake `id` based on the current time and uses that `id` to add the new `contestant` to the `contestants` collection. I would not recommend using this method in a real application, but it will serve its purpose until it gets hooked up to Firebase.

```
service.addContestant = function (contestant) {
  contestant.id = new Date().getTime();
  contestants.push(contestant);
};
```



Race to Awesome

Lane	Contestant	Score	
4	Speedy Gonzalez	6	Remove
5	Lukas Ruebbelke	4	Remove

Add Contestant

Race to Awesome

Lane	Contestant	Score
4	Speedy Gonzalez	6
5	Lukas Ruebbelke	4

Leaderboard When a user updates a contestant in the dashboard view, that change is immediately sent to everyone connected to the leaderboard view

Updating a contestant

Updating a contestant is slightly simpler in the static version of the leaderboard, as updates to the contestant object are immediately stored in memory. In the dashboard, we will replace the simple binding in the table cell to a text field that we will bind to that property on the `contestant`.

For instance, the markup below:

```
<td>{{contestant.lane}}</td>
```

... will become:

```
<td><input type="text"
  ng-model="contestant.lane"
  ng-blur="main.updateContestant(contestant)"/></td>
```

We are using `ngBlur` to call `main.updateContestant` with the `contestant` we are updating, which then gets passed on to `ContestantsService.updateContestant`.

```
main.updateContestant = function (contestant) {
  ContestantsService.updateContestant(contestant);
};
```

We are not actually going to do anything in `service.updateContestant` yet, but it will come in handy in just a moment. ▶

FOCUS ON

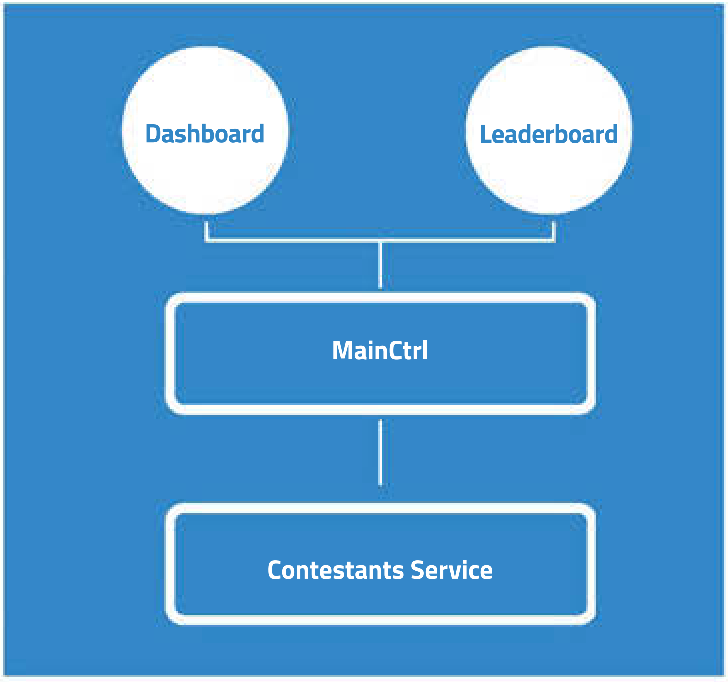
ANGULARFIRE CHEATSHEET

AngularFire is the officially supported Firebase library that allows us to bind an AngularJS collection to a Firebase endpoint. AngularFire enables our AngularJS collections to be seamlessly kept in sync with the Firebase services. It also provides an easy to use API for modifying the remote collection so that all other connected clients are updated.

- **Read items** `= $firebase(new Firebase(FIREBASE_URI)).$asArray()`
- **Create** `items.$add(item)`
- **Update** `items.$save(item)`
- **Delete** `items.$remove(item)`

Once an AngularJS collection has been initialised with the `$firebase` service, there is an implicit read relationship in that the collection is automatically kept in sync with the Firebase servers. To modify the collection, we must explicitly perform those operations using the AngularFire API, which allows us more granular control over how we update the servers.

You can read more about the library at *netm.ag/cheatsheet-265*.



The big picture In this application, we have a dashboard and leaderboard view that uses the MainCtrl, which gets contestants data from the ContestantsService

```
service.updateContestant = function (contestant) {
  // Already in memory
};
```

Removing a contestant

The final piece of the application we need to talk about is the ability to remove a contestant. In the dashboard, we have an extra column that we will use to add a button that calls `main.removeContestant` with the `contestant` we want to delete.

```
<button type="button" class="btn btn-link"
  ng-click="main.removeContestant(contestant)">Remo
ve</button>
```

The `main.removeContestant` method then passes that contestant on to the `ContestantsService.removeContestant` method.

```
main.removeContestant = function (contestant) {
  ContestantsService.removeContestant(contestant);
};
```

> We can now use $firebase to fetch a real-time collection of our contestants

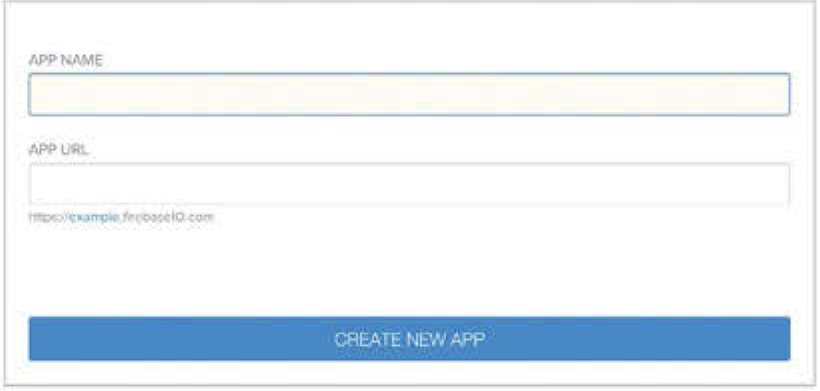
This is `service.removeContestant` uses the `contestant.id` property to remove it from the collection.

```
service.removeContestant = function (contestant) {
  contestants.remove(function(c) { // Sugar.js method call
    return c.id === contestant.id;
  });
};
```

We have now laid the foundation for the AngularJS portion of the application – this is where the fun begins! In the next few moments, we are going to see how easy it is to convert our static leaderboard, with mock data and service methods, into an application with real data that updates in real time.

Setting up a Firebase account

The very first step to creating a Firebase application is setting up a free Firebase account at *firebase.com*. Next, we need to create a Firebase app instance for our application. From the account page, this will be the first form you see. You just need to give your app a name and hit 'Submit' to generate your example



New application To create a new Firebase application, simply enter in a new application name in the Firebase dashboard and hit 'Create new app'

and generate a URL based on that application name. Next, we need to add the appropriate resources to our new project.

We will add in the base `firebase.js` file and the `angularfire.min.js` library to our leaderboard. The `firebase.js` file is responsible for managing the synchronisation between the client and server and AngularFire is the official support library for binding an AngularJS app to a Firebase collection (see boxout opposite).

```
<script src="//cdn.firebase.com/js/client/2.1.0/firebase.
js"></script>
<script src="//cdn.firebase.com/libs/angularfire/0.9.1/
angularfire.min.js"></script>
```

With the appropriate resources added, we need to add the `firebase` submodule to our application.

```
var app = angular.module('leaderboard', ['firebase']);
```

One other thing that I like to do is extract that Firebase endpoint into a constant so that it can be defined once and referenced throughout the entire application.

```
app.constant('FIREBASE_URI', 'PUT_YOUR_FIREBASE_
HERE');
```

REAL-TIME SERVICE

The stage is now set for us to convert `ContestantsService` from a static service into a mechanism with real time capabilities. The impressive part is that we will have spent significantly more time establishing context than actually wiring the leaderboard up to Firebase.

We are going to replace the static `contestants` collection with a real-time collection that lives in the Firebase servers.

To do this, we need to inject the `$firebase` service as well as the `FIREBASE_URI` constant we created.

```
app.factory('ContestantsService', function ($firebase,
FIREBASE_URI)
```

Within the service, we are going to initialise a new Firebase reference using the endpoint to the Firebase application we just created.

```
var ref = new Firebase(FIREBASE_URI);
var contestants = $firebase(ref).$asArray();
```

We can now use the `$firebase` service to fetch a real-time collection of our contestants by passing in the Firebase `ref` we just created. The `$asArray` method returns a read-only array that we can use with `ngRepeat` . We can return our `contestants` collection just as before, and the leaderboard will continue to work!

```
service.getContestants = function () {
  return contestants;
};
```

Adding a contestant

Using `$asArray` gives us an array to bind to, but we need to use special Firebase methods if we want to manipulate the collection.

To add a new contestant, we call `$add` on the `contestants` collection and pass in the new `contestant` .

```
service.addContestant = function (contestant) {
  contestants.$add(contestant);
};
```

Updating a contestant

Updating a contestant works almost in an almost identical way to adding a contestant, except we call `$save` instead of `$add` .

```
service.updateContestant = function (contestant) {
  contestants.$save(contestant);
};
```

Removing a contestant

This may come as a big surprise, but removing contestant is just like the adding and updating a contestant, except we use the `$remove` method and pass in the contestant we want to remove.

```
service.removeContestant = function (contestant) {
  contestants.$remove(contestant);
};
```

Did we really just give a regular AngularJS application real-time capabilities in less than 10 lines of code? Yeah we did! ◼

RESOURCE

REMOTE MOBILE

Lukas Ruebbelke has created an additional view for a mobile device. It works as a remote, enabling users to alter an athlete's score on the go. Find it in the GitHub repo at *netm.ag/firebasegit-265*

DISCOVER NET MAGAZINE

The voice of web design
net
THE TOP 20 WEB DESIGN TRENDS 2015
Leading pros reveal the skills you need this year
BUILD THE PERFECT WEBSITE
Future

SAVE UP TO

45%

GO DIGITAL

The voice of web design
net
PRO IMAGE OPTIMISATION TECHNIQUES
TOP 10 WEB DESIGN TOOLS For 2015
We review the hottest design tools for 2015

#263 FEB 2015

Discover the design tools set to revolutionise your workflow this year, plus we look at how to boost site speed by

The voice of web design
net
DISCOVER THE POWER OF SVG EFFECTS
BUILD MOBILE APPS THE EASY WAY

#262 JAN 2015

We reveal everything you need to know to build your first mobile app, and uncover the power of SVGs

#264 MARCH 2015

We present the techniques you need to build your best ever site. Plus, a look at the top 20 emerging design trends

FROM

£20.49

Based on a 6-month subscription

GOT AN APPLE DEVICE?

Download **net**'s interactive iPad edition, or read a print replica on your iPhone or iPod Touch

NETM.AG/NET-IPAD (UK), NETM.AG/NET-IPAD-US (US) Available on the App Store

NEVER MISS AN ISSUE! CATCH UP BY DOWNLOADING OUR DIGITAL EDITIONS

#261 DEC 2014

Get the low-down on the top JavaScript libraries. Plus, find out how to build a modular CMS in WordPress

#260 NOV 2014

Explore the advanced techniques that will help you build responsive sites that work seamlessly on any device

#259 OCT 2014

Find out how to build native apps with Steroids, and explore the exciting new features in WordPress 4.0

#258 SEPT 2014

We share the SEO tips that will propel your sites to the top of Google, and show you how to mock-up a site with Sketch 3

#257 SUMMER 2014

Want to join the Sass revolution? We show you how to make the most of this popular preprocessor

#256 AUG 2014

We reveal 12 mind-blowing HTML5 hacks, and walk through how to create code-free designs using Macaw

net
GET STARTED WITH GOOGLE APP ENGINE
THE Complete WEB DESIGN TOOLKIT

#255 JULY 2014

Find out how to take the pain out of frontend development with our handy Web Design Toolkit feature

#254 JUNE 2014

Discover how you can be your own boss with our designer's guide to going freelance. Plus, we explore the Web Audio API

PREFER TO READ ON ANDROID, PC OR MAC?

net is also available on Google Play and Zinio, as well as Kindle, Nook, Windows 8 and more.

NETM.AG/NET-GOOGLEPLAY GET THE MAGAZINE ON Google play NETM.AG/NET-ZINIO zinio™

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Download the files here!

Download the GitHub repository for this tutorial at *netm.ag/ionicgit-260*

ABOUT THE AUTHOR

JAMES MILLER

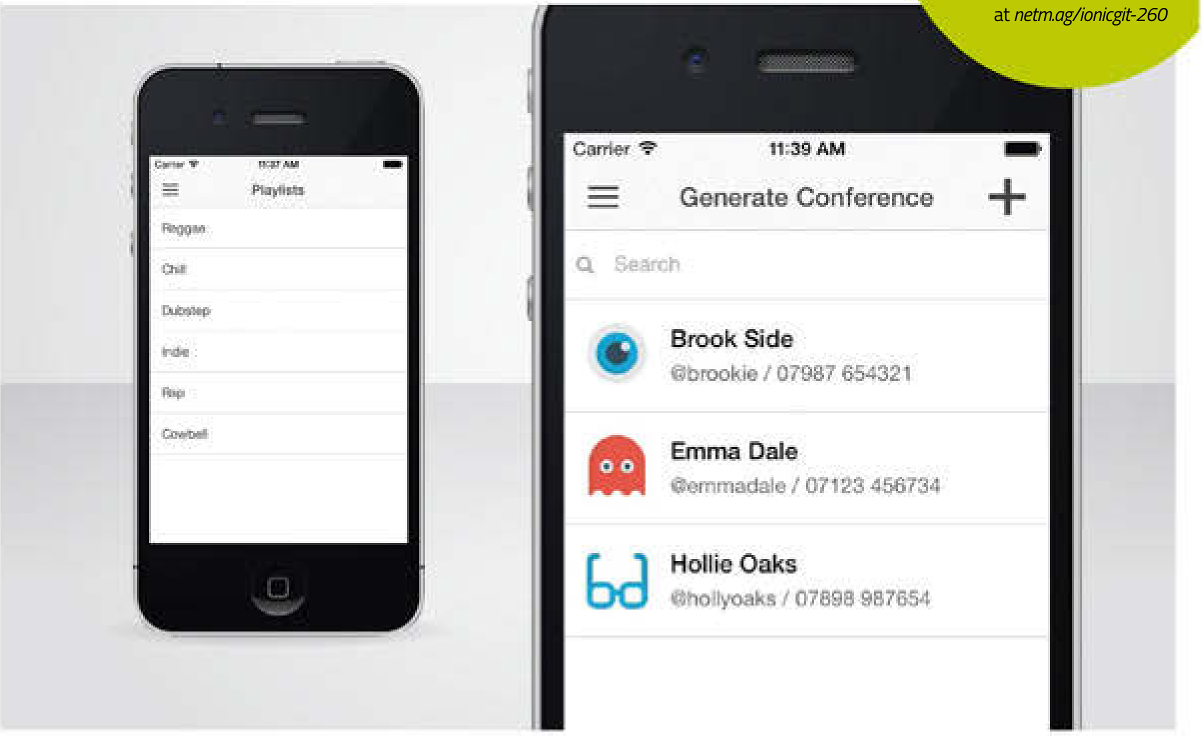
w: *james-miller.co.uk*

t: @jimhunty

job: Technical lead, AnalogFolk

areas of expertise: Frontend development

q: what was your childhood nickname?
Hunty – it was my fantasy wrestler's name on my first ever web project



MOBILE

BUILD HYBRID MOBILE APPS WITH IONIC

The Ionic framework enables web developers to create mobile apps in a rapid development environment. **James Miller** explains how

I like frontend development. It's simple and quick. In comparison, creating apps for mobile platforms can be long and laborious. Thankfully, programmers have brought the world of native apps to web developers through HTML5 hybrid app frameworks. These use the device's webview to display the app, so we can use frontend technologies like HTML, CSS and JavaScript to build apps that give the same experience. Using a framework means one set of code can be deployed onto multiple platforms, reaching a wide audience with minimum effort.

Ionic (*ionicframework.com*) is a rapid development framework for hybrid mobile apps. It is made up of three different technologies:

- Cordova (*cordova.apache.org*) forms the core of the application framework, allowing access to native phone functionality through the use of JavaScript. Cordova is the open source version of PhoneGap, an HTML5 hybrid app framework
- AngularJS (*angularjs.org*) is Google's JavaScript MVW framework, supplying the business logic part of the app. Angular provides a way to structure JavaScript to make use of modular patterns and correct separation of concerns
- Ionic's UI framework is a set of tools that help you build aesthetically pleasing, native user interfaces. Similar to Bootstrap, it can be used to write HTML in its prescriptive manner, instantly delivering a functional interface

I have put together a networking app for **net**'s Generate conference (*netm.ag/ionicgit-260*) to go alongside this guide. There is significantly more

VIDEO

James Miller has put together an exclusive screencast to go alongside this tutorial. To watch along, visit *netm.ag/ionicvid-260* detail in the app than the guide touches on, so have a rummage if you're interested in exploring further.

First, follow the installation instructions at *netm.ag/gettingstarted-260*. Once completed, you will have installed Ionic, Cordova, and will be set up for development on Android (and iOS if you're on a Mac).

COMMAND LINE INTERFACE

For frontend developers the command line can sometimes be a scary place, but with the emergence of technologies like npm, Grunt and Gulp, we've really got to grips with it. The Ionic command line interface (CLI) is built on top of Cordova's; a natural extension that uses the same intuitive commands, as well as a few extras.

To get started, open up the terminal and `cd` to the folder that you wish to build the app. Type in the lines below, pressing enter after each:

```
$ ionic start Generate sidebar
$ cd Generate
$ ionic platform add ios
$ ionic build ios
$ ionic emulate ios
```

HTML5 hybrid app frameworks bring the world of native apps to web developers

So what do those lines of code do? The first creates an Ionic build from a sidebar menu template, including Cordova, AngularJS and the UI framework. The next moves us into the newly created folder, where the third adds iOS as a platform (alternatively you could use Android, by replacing the references to `ios` with `android`). The fourth line runs the build, which moves the top level of files to the platform level. The final line runs the iOS simulator, which should show everything correctly installed.

Creating the project also creates a demo application, so you can play around with Ionic's features straight away.

THE INTERFACE

When building an app in Ionic, you must only work within the top-level `www` folder. The files within this folder are used to build the application.

In `/templates` you will find HTML files with snippets of code that are injected into the application. The aim here is reusability, giving a more modular approach to development.
In the sample app, I renamed the templates ▶

RESOURCES

ANGULAR 101

You will find Angular's steep learning curve a lot easier to scale if you have a good grip of the terminology used.

Two-way data binding is described as Angular's killer feature for a reason. It allows the data structure to be combined with elements on the page. If you change a field in a form bound to another element – for example a heading – as you type, the changes would appear instantly in the heading. This forms part of the scope.

Scopes are JavaScript objects that allow the division of an application through inheritance. Each object will have its own scope made up of its own set of variables and functions, however it can also inherit these from its parents. Scopes allow for concerns to be better laid out and provide a more focused approach.

Directives create custom HTML elements and define their interactivity. This mixes templating and JavaScript functionality to create reusable components. Out of the box, Angular offers a number of ready-to-use directives.

Views are used to display the user interface. They consist of HTML, CSS, templating and directives.

Services organise and share data. Use services to connect to an external server and to manage the data layer of the application.

Controllers manage the user interactions that bind your view and service.

Modules collect and organise components (scopes, directives, views, services and controllers, plus others) allowing you to abstract areas of the application.

Learning resources

- **Official Angular JS tutorial** (*docs.angularjs.org/tutorial/step_00*)
  Start here for the fundamentals, including correct unit testing through Karma
- **Egghead.io** (*egghead.io*)
  John Lindquist has a tremendous set of video tutorials featuring all aspects of AngularJS
- **Angular best practices** (*netm.ag/best-260*)
  Joe Eames' videos on PluralSight offer excellent explanation on structure and the best practice ways to use this framework

and their assigned JavaScript functionality to maintain a working application throughout development. To display a list of contacts on the `network.html` template, a few Angular directives were used.

```
<ion-list>
<ion-item class="item-avatar" ng-repeat="person in network
| orderBy:'name' | filter:search " href="#/app/person/{{person.
id}}"> ...</ion-item>
</ion-list>
```

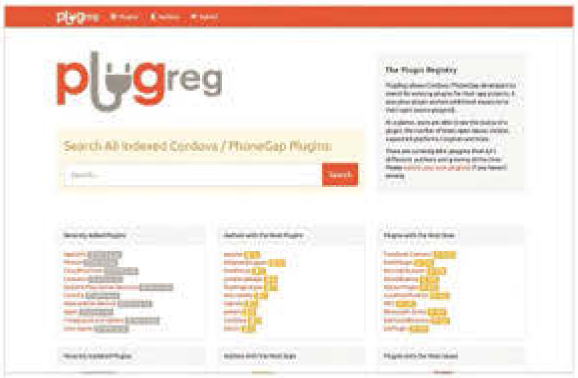
The code uses the elements created by Ionic's custom directives. This provides all the styling and functionality needed to make things look great without much effort.

`<ion-list>` creates a placeholder for the list, and using `ng-repeat` creates a loop on `<ion-item>`. Identical to a `foreach` loop, this lists each contact that exists in the network array (obtained from a service) and displays their data.

When something is typed in the search box, this code will remove items that don't match

On the directive `ng-repeat`, a `filter:search` is specified. The filter points to a text input box that sits above the list and acts as a search. When something is typed in the search box, this simple piece of code will remove items from the list that don't match. Before Angular this kind of functionality required much more code, but now is significantly easier to implement.

The file `js/app.js` initialises the app, setting up the top-level functions for the application. URLs decide which template (page) is injected in to the



Plugreg An excellent resource, linking browsing plugins to their sources

FOCUS ON

MOBILE EMULATION WITH CHROME

When building apps with Ionic, the default method of testing is to emulate the device by using the command line interface. Waiting for Ionic to compile, build and install the app on the emulator can be a slow and painful process. Fortunately Chrome offers an alternative.

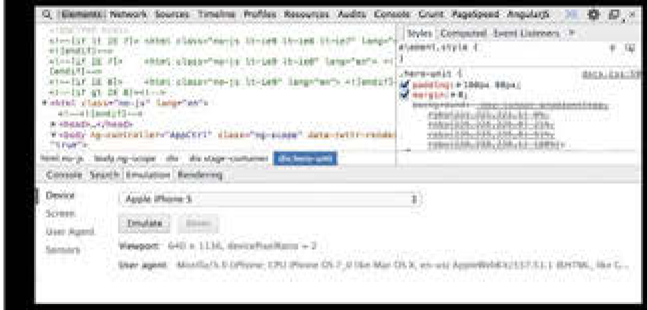
Chrome developer tools include the ability to emulate a mobile device (*netm.ag/emulation-260*). As Chrome uses the same browser engine (Webkit) as the webview in Ionic, there should be like-for-like emulation. To enable mobile emulation, navigate to your project in Chrome. Right-click and select Inspect Element to bring up the developer tools. Next choose the blue symbol at the top right of the Inspect Element window – it looks like a '>' symbol with three horizontal lines. This will display a secondary menu at the bottom of the screen, where you should choose the Emulation tab.

Choose your desired device from the dropdown and press Emulate. You may need to refresh the page to start the emulator.

Features:

- **Screen** Viewport resolution adjusts the window view, setting the boundaries to the device chosen. You can also change the orientation, emulating both landscape and portrait
- **Sensors** Touch Events replaces mouse clicks with emulated touch allowing you to test the performance of gestures
- **User Agent** Spoofing tells Chrome you are coming from a mobile device, rather than the desktop that you are developing with

The obvious downfall to this approach of emulation is that you can't use the native plugins (camera, contacts and so on) available within Ionic as there is no physical device. Likewise Chrome will be more powerful than the device's webview, so performance when emulating can be misleading. For functionality and styling however, Chrome is extremely helpful at speeding up development of HTML5 hybrid apps.



Device: Apple iPhone 5

Viewport: 640 × 1136, devicePixelRatio = 2

User agent: Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Mac OS X; en-us) AppleWebKit/537.51.1 (KHTML, like G...

Emulation The device emulation is fairly hidden – this is where you will find it application and how it will be controlled. Below is the example application's person page in the router.

```
.state('app.person', {
url: "/person/:personId",
views: {
'menuContent' :{
 templateUrl: "templates/person.html",
 controller: 'PersonCtrl'
} } } )
```

The purpose of the page is to display the contact's data in a form. At the end of the `url` line `:personId` is defined. This is a namespace for a variable (number or string) that can be parsed through to the Controller. Once received by the Controller (see next section), it can talk to the data store and obtain the data relating to that contact's details.

The page's `templateUrl` links to the template file `person.html`, which contains all of the HTML required to display the page. Finally, we define the Controller that will manage the interactions for the page.

CONTROLLERS

Controllers (`js/controllers.js`) manage user interactions as well as the discussion between data store and user interface. Within a Controller you can manage what dependencies are injected, ensuring each area of the app gets exactly what it needs.

```
.controller('NetworkCtrl', function($scope, $location,
PeopleService){
  $scope.network = PeopleService.list();
... })
```

Here the Controller manages the network page. Note the `PeopleService` dependency, which is our gateway to the data storage. A variable called network is set to access the `PeopleService` to get a list of contacts to display. This is how the data goes from end to end.

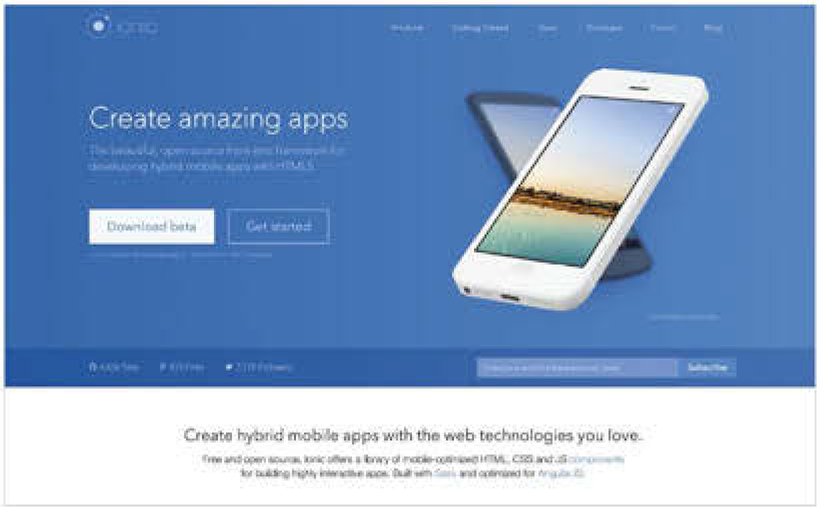
SERVICES AND STORAGE

There are different ways to write services (Providers, Factories, Service and so on) but the differences are negligible at this level. For a small app, a standard service will do the job, as with `PeopleService`.

Within this service is the `list` function:

```
this.list = function(){
   var people = JSON.parse(localStorage.getItem('people'));
   return people; }
```

We are using `localStorage` as a permanent data store. The list function retrieves the JSON object from `localStorage` and returns it for use within the controller. `localStorage` is perfect here, as it creates a simple, device-independent store. If your project gets bigger, consider Cordova's storage plugin.



Create amazing apps

Download beta

Get started

Create hybrid mobile apps with the web technologies you love.

Free and open source, Ionic offers a library of mobile-optimized HTML, CSS and JS components for building highly interactive apps. Built with Sass and optimized for AngularJS.

Helpful info Ionic's website is a constantly growing resource of information for each of the technologies it uses

EXTENSION THROUGH PLUGINS

With Cordova at its core, Ionic has the option to use Cordova's official and community plugins. The Cordova community has created plugins to cover almost all native functionality, accessible through JavaScript. The most complete source of plugins available is PlugReg (*plugreg.com*).

Plugins are installed using the CLI. The example app uses the official notifications plugin. To install the notifications plugin, type:

```
$ cordova plugin add org.apache.cordova.dialogs
```

This can be seen in action when a user wants to delete a contact. This makes it easy to display a native confirmation pop-up, instead of a typical JavaScript one.

```
navigator.notification.confirm(
 'Are you sure?', // popup message
 $scope.removeContact, // callback function
 'Delete Contact', // box title
 ['Delete','Cancel'] // button labels
);
```

If you install plugins, remember to build before you emulate to make sure the plugin files are distributed into the correct platform-specific folders.

CONCLUSION

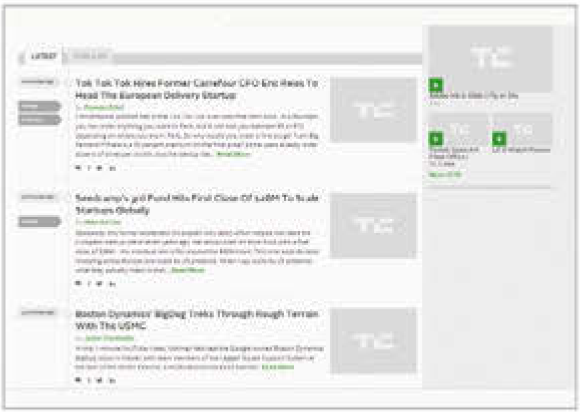
Beyond these basics, you can extend the Sass file to give your app a more customised look and feel, add more JavaScript components, and even deploy the finished product on the app stores.

RESOURCE

IONIC CREATOR

The Ionic team are putting together a drag-and-drop interface, called Ionic Creator to help non-developers create Ionic apps. Looks a very exciting project to follow. *ionicframework.com/creator*

PERFORMANCE & WORKFLOW



LATEST

Tok Tok Tok Hires Former Carrefour CFO Eric Reiss To Head The European Delivery Startup

Seedcamp's 3rd Fund Hits First Close Of $28M To Scale Startups Globally

Boston Dynamics' BigDog Treks Through Rough Terrain With The USMC

MY REPORTS | OCTOBER NEWSLETTER

2,301	12	8	408

Summary

ABOUT THE AUTHOR
DEN ODELL
w: *akqa.com*
t: @denodell
areas of expertise: JavaScript, CSS3, semantic markup, code management
q: what's the coolest thing you do offline?
a: I DJ in clubs around Europe. I once played the warm-up set for Soulwax and 2ManyDJs in front of 20,000 festival goers

*PROCESS

SEVEN STEPS TO BETTER JAVASCRIPT

Den Odell presents his seven-step plan for writing flawless code, and rounds up the most useful tools for streamlining the process

With browser performance improving, along with the steady adoption of new HTML5 APIs, the volume of JavaScript on the web is growing. Yet a single poorly written line of code has the potential to break an entire website, frustrating users and driving away potential customers.

Developers must use all the tools and techniques at their disposal to improve the quality of their code to be confident that it can be trusted to execute predictably every time. This is a topic close to my heart and I've been working over many years to find a set of steps to follow during development to ensure only the highest-quality code gets released.

Follow these seven steps to dramatically improve the quality of your JavaScript projects. With this workflow, fewer errors will occur and any that do will be handled gracefully, leaving users to browse without frustration.

1 CODE

Start by invoking ECMAScript 5's strict mode (*netm.ag/strictmode-249*) in your functions with a `"use strict"` statement, and use the module design pattern (*netm.ag/modulepattern-249*), minimising the use of global variables by sandboxing separate code modules within self-executing function closures, passing in any external dependencies to keep modules clear and concise. Only use established and well-tested third-party libraries and frameworks, and keep your functions small, separating any business logic or data from your DOM manipulation and other view-layer code.

Larger projects with multiple developers should follow an established set of coding guidelines, such as Google's JavaScript Style Guide (*netm.ag/jsguide-249*), and will need stronger code management rules, including stricter dependency management using the Asynchronous Module Definition (AMD) through a library such as RequireJS (*requirejs.org*), package management using Bower (*github.com/bower/bower*) or Jam (*jamjs.org*) to reference specific versions of your dependency files, and the use of structural design patterns, such as the Observer pattern (*netm.ag/observer-249*), to facilitate loosely-coupled communication between your different code modules.

It's also a wise idea to use a code storage repository system such as Git or Subversion via services such as GitHub (*github.com*) or Beanstalk (*beanstalkapp.com*) to keep your code backed up in the cloud, provide the ability to revert to previous versions, and, for more advanced projects, to create branches of code for implementing different features before merging them back together once complete.

SLIDEDECK

MORE DEPTH

The slides for Den Odell's conference presentation 'High-quality JavaScript Code' expands on the ideas set out in this article: *netm.ag/quality-249*

2 DOCUMENT

Use a structured block comment format such as YUIDoc (*yui.github.io/yuidoc*) or JSDoc (*usejsdoc.org*) to document functions so any developer can understand its purpose without needing to study its code, reducing misunderstanding. Use Markdown syntax (*netm.ag/markdown-249*) for richer, long-form comments and descriptions. Use associated command-line tools to auto-generate a documentation website based on these structured comments that keeps up to date with any changes made in your code.

3 ANALYSE

Run a static code-analysis tool, such as JSHint (*jshint.com*) or JSLint (*jslint.com*) against your code regularly. These check for known coding pitfalls and potential errors, such as forgetting to use strict mode or referencing undeclared variables, and spot missing braces or semicolons. Correct any issues the tool flags up to improve your code quality. Try setting default options for your project team to enforce coding standards, such as the number of spaces by which to indent each line, where to place curly braces, and the use of single or double quotes throughout your code files.

With these seven steps, fewer errors will occur and those that do will be handled gracefully

4 TEST

A unit test is a small standalone function that executes one of the functions from your main codebase with specific inputs to confirm it outputs an expected value. To improve your confidence that code will behave as expected, write unit tests using a framework such as Jasmine (*pivotal.github.io/jasmine*) or QUnit (*qunitjs.com*) for each of your functions, using both expected and unexpected input parameters. And don't forget those edge cases!

Run these tests in multiple browsers across multiple operating systems by taking advantage of services such as BrowserStack (*browserstack.com*) or Sauce Labs (*saucelabs.com*) which allow you to spin up virtual machines in the cloud on demand for testing. Both services provide an API allowing your unit tests to be run automatically across a number of browsers simultaneously, with the results fed back to you as they complete.

5 MEASURE

Code-coverage tools such as Istanbul (*gotwarlost.github.io/istanbul*) measure which lines of code are executed when your unit tests run against your functions, reporting this as a percentage of the total number of lines of code. Run a code-coverage tool against your unit tests, and add extra tests to increase your coverage score to 100 per cent, giving you greater confidence in your code.

Function complexity can be measured using Halstead complexity measures (*netm.ag/halstead-249*): equations devised by computer scientist Maurice Halstead in the 1970s that quantify the complexity of a function according to the number of loops, branches and function calls it contains. As this complexity score decreases, the easier it becomes to understand and maintain the function, reducing the likelihood of errors.

The command-line tool Plato (*github.com/es-analysis/plato*) measures and generates visualisations of JavaScript code complexity. It helps you identify functions that could be improved, while storing previous results, allowing quality progress to be tracked over time.

6 AUTOMATE

Use a task runner such as Grunt (*gruntjs.com*) or Gulp (*gulpjs.com*) to automate the processes of documentation, analysis, testing, coverage and complexity-report generation, saving yourself time and effort, and increasing the chance of addressing any quality issues that arise. Most of the tools and testing frameworks highlighted in this article have associated Grunt and Gulp tasks available to help you improve your workflow and your code quality without you having to lift a finger.

7 HANDLE EXCEPTIONS

Invariably, at some point, your code will throw an error when it's run. Use `try...catch` statements (*netm.ag/trycatch-249*) to handle runtime errors gracefully and limit impact on your site's behaviour. Use a web service to log runtime errors thrown. Use this information to add new unit tests so as to improve your code and eradicate these errors one by one.

STEPS TO SUCCESS

These seven steps have helped me produce some of the code I'm most proud of in my career so far. They're a great foundation for the future, too. Commit to using these steps in your own projects to produce high-quality JavaScript code, and we can work together to improve the web, step by step.

RESOURCE

USEFUL BOOK

Published by Apress Media, Den Odell's 2009 book, *Pro JavaScript RIA Techniques*, outlines best practices for writing code: *netm.ag/projavascript-249*

VIDEO

Watch an exclusive screencast of this tutorial created by Tuts+ Premium: *netm.ag/tut2-249*

ABOUT THE AUTHOR
NICHOLAS ZAKAS

w: *nczonline.net*

t: @slicknet

areas of expertise:
HTML5, CSS3, JavaScript, performance, architecture

q: what's the oddest thing you've ever seen on a designer's desk?
a: A Care Bear

PERFORMANCE

THE FOUR JAVASCRIPT LOADING TIMES

Nicholas Zakas presents four JavaScript loading times to help improve your web application's overall performance

More and more it seems that web applications are running into performance problems that can often be traced back to JavaScript. It also seems we're having more problems with performance now despite JavaScript engines being much faster today compared to five years ago. Perhaps older web applications benefited from a population with slower internet connections, or maybe no one thought that much about performance back then. It's easy to see why we're having problems now.

Every leap in computing power tends to be accompanied by a leap in the amount of computing power used by software. Software capabilities always grow to fill the processor and memory limits available. Web applications today use much more JavaScript than they used to, partially because JavaScript code is executed so much faster. The same code that could take two whole seconds in Internet Explorer 7 can take less than a millisecond in Chrome. As a result, web applications keep adding more and more JavaScript, inextricably tying the application's experience to the JavaScript.

It was common in my consulting practice to find web applications with over 1MB of required JavaScript (minified and un-gzipped). My first task was always to identify and eliminate any unnecessary or redundant JavaScript. After that, left with the bare minimum (often still close to 1MB), I'd move on to the next phase: splitting up the various pieces of JavaScript by when they were loaded. This is when I developed the concept of the four JavaScript loading times.

The four JavaScript loading times is a mental model and a technical solution for using JavaScript in a web application. The mental model asks some questions about your JavaScript usage, such as when certain components are actually needed and what trigger tells you of their necessity. The technical solution ensures the best possible user experience by providing points in time at which it's appropriate to load those JavaScript components. Perhaps the most important aspect of the four JavaScript loading times is in their ability to help you better understand your users' behaviour.

LOADING TIME 1: IN THE <HEAD>

The first JavaScript loading time is in the `<head>` element of a page. Before web performance engineer Steve Souders informed the web development community that this wasn't a good idea, most web applications included JavaScript files at the top of the page along with CSS style sheets. JavaScript loaded in the `<head>` prevents the document body rendering, which results in a blank page being displayed to the user while the browser waits to download and execute the JavaScript. As such, most web applications no longer load JavaScript in the `<head>`. Unfortunately, there's a whole class of JavaScript

SLIDEDECK

See Nicholas Zakas's presentation 'Enough with the JavaScript Already!' on JavaScript usage, which this tutorial is based on: *slideshare.net/nzakas/enough-withthejavascript already* that must be loaded early in the page in order to function correctly. Many types of analytics software that use JavaScript libraries to track user actions need to be loaded as early as possible in order to accurately track everything that happens during the lifecycle of the page. This may be for user-testing or advertising purposes, but the end result is that this code must be loaded in the `<head>`.

You can't completely forget about this first JavaScript loading time because of the type of software that requires it. Analytics are very important for web applications and so it's not possible or prudent to wait to load these libraries at a later time. In fact, it's the analytics software that makes it possible to determine what should be loaded in the other JavaScript loading times.

LOADING TIME 2: BOTTOM OF <BODY>

Souders's recommendation is to load all JavaScript at the bottom of the `<body>` of a page, just before the `</body>` tag. Doing so allows the page to render completely before starting to download and execute JavaScript. This loading time is the best practice for getting JavaScript onto a page while it's being loaded. If you use progressive enhancement, users are still able to interact with the page even before the JavaScript finishes loading. Doing so allows the time to interactivity (the time between the user requesting the page and being able to complete an action) to be as fast as possible.

> It's the analytics software that makes it possible to determine what should be loaded

This is the area in which you should include all JavaScript that's necessary to be on the page at the page load time. What that typically means is any foundational JavaScript libraries (like jQuery, YUI, Backbone) plus any bootstrap code that prepares the application to be used. You may also want to load the JavaScript for whatever is the most common task that a user does immediately after page load.

Here are some examples:

- **Email application** You may want to load code for reading an email as well as the email list
- **Online store** You may want to load any code related to search
- **File sharing application** You may want to load the code related to downloading and uploading

▶

FOCUS ON

LOAD TIMES FOR RETAILER SITES

Think of a typical product page on a shopping site. When you first go to the site, chances are you search for something and then click through on a search results page to the product page. Once on the product page, you're likely to read the description of the product and look at the photo to see if it's actually what you want. Assuming it's the correct product then you're likely to add the item to your shopping cart. You can surmise that the most important functionality on a product page is the button to 'add to cart' rather than to the search box at the top of the page or other options lower on the page. In this case, you would load JavaScript for the 'add to cart' button in the second loading time (before `</body>`) and defer the search suggestion JavaScript until the third or fourth time, depending on what your analytics shows you.

Practice using this technique by monitoring how you interact with all the different sites you visit on a day-to-day basis. Keep a note of the top three things you do on any one page. For example, when you log into Facebook, do you do so to post or to just read something someone else has posted? How often do you do a search on Amazon? Do you tend to tweet from Twitter, or mostly read what other people have posted? How often do you scroll when visiting any site? Keeping track of your own browsing habits can help get you into the mindset of users to figure out when to load what functionality.

amazon
Off to College
TEXTBOOKS: RENT. BUY. SELL.
Professional JavaScript for Web Developers
JavaScript for Web Developers
Add to Cart

Shopping cart Consider how you use web pages to help set your JS loading times

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

The key to this loading time is to figure out the minimum amount of JavaScript code you need in order for the user to be successful with the first interaction on the application. All of the JavaScript code to this point prevents the firing of the `load` event. The browser will appear to still be busy while the JavaScript code is being downloaded and executed. Once the `load` event is fired, the browser no longer appears busy and visually appears to be loaded. The goal is always to make the transition from loading to loaded as quickly as possible.

LOADING TIME 3: AFTER LOAD

The third JavaScript loading time occurs after the `load` event. This is the ideal time to start loading in additional JavaScript code that may be necessary for other tasks the user may do during the lifecycle of the application. For instance, if your email application also offers chat, you may want to start downloading the chat code at this time. Since chat is a secondary feature in that case, it's OK to slightly delay the downloading of this code.

The possibilities are vast for detecting and responding to user behaviour

JavaScript code downloaded during this loading time is accomplished using the `onload` event handler and dynamic script loading.

A simple example looks like this:

```
  window.addEventListener("load", function() {
    var script = document.createElement("script");
    script.src = "/more/code.js";
    document.body.insertBefore(script, document.body.
firstChild);
  });
```

We create a `<script>` element using the Document Object Model (DOM) and then add it to the page. This example uses a single file but you can load as many additional files as you would like. As the order of execution when loading multiple files isn't guaranteed, it's best if each file can live on its own.

This loading time is very effective when used with a progressively enhanced site as you can continue to load in more dynamic functionality, even as the page is already capable of serving the user needs. Large web applications frequently take advantage of this loading time.

LOADING TIME 4: ON DEMAND

On demand JavaScript loading can literally happen at any point in time based on how the user is interacting with the web application. It could be as simple as waiting until focus is set into the search box before loading search functionality, or watching for the mouse cursor to move towards an image before loading the zoomed-in view code. You can also make it more complex by inferring what the user will do next based on the previous couple of actions. There are endless opportunities to get creative when loading JavaScript on demand, which makes it a very powerful option.

Here are some options:

- Load JavaScript for search suggestions only once when the user sets the focus to the search box
- Load JavaScript for a button only when the user moves the mouse within 200 pixels of it
- Load JavaScript for the page footer only when the user scrolls down the page
- Load JavaScript for image manipulation only when the user moves the mouse over an image

With the large number of JavaScript events available, the possibilities are vast for detecting and responding to user behaviour. Unfortunately, most web applications fail to take advantage of the fourth JavaScript loading time.

It doesn't take a lot of extra work but it can significantly reduce the amount of JavaScript code you need to download every time somebody loads the application. All it takes is a little bit of thought about how people are using the application and a little bit of code (of course, analytics helps as well).

USING THE FOUR LOADING TIMES

While working with my clients, I encourage them to think of JavaScript functionality in terms of the four loading times and consider what has to appear immediately. What needs to be there before the `load` event? What has to be there after? What can we load based on what the user is doing?

Based on the four loading times, you can easily devise a plan for when each type of JavaScript has to be added onto the page. This plan is based on how users interact with your web application. Let your application users tell you what functionality is absolutely critical and at what point in time it will first be needed.

Finally, user analytics are often the missing piece to an important web performance puzzle, and the four JavaScript loading times use this data to make informed decisions about JavaScript that, in turn, will improve your app's performance.

SLIDEDECK

See Steve Souders's presentation, 'JavaScript Performance' from the San Francisco JavaScript meetup *slideshare.net/souders/javascript-performance-at-sfjs*

MOBILE

SPEEDIER PERFORMANCE

Aaron Gustafson on the new techniques set to speed up mobile experiences

As web designers, we're in a bit of a pickle when it comes to content delivery – especially over mobile networks. These tend to be plagued by high latency, which can increase download times, while users have an expectation that sites should load just as fast on these devices. Sadly, both of these issues are completely out of our control. That said, the W3C has been continuing to explore ways we can craft our markup to control how assets are downloaded and when they are executed in the browser.

NOW PLAYING: SCRIPT CONTROLS

For a while now, the common wisdom has been to move all `script` elements to the bottom of the `body` element. This is because when browsers encounter a `script` element, rendering stops until the JavaScript source is downloaded and executed – just in case that script causes the page to render differently. This is known as JavaScript blocking.

As part of the HTML5 spec, two attributes were introduced to allow control over when and how JavaScript is executed: `defer` and `async`. The `defer` attribute tells the browser to wait to download and execute the JavaScript code until it has finished processing the document. The `async` attribute tells the browser that the JavaScript file in question can be safely executed asynchronously (i.e. while the browser continues doing other things).

I would be remiss if I did not mention one potential issue with using these attributes: dependency loading. If two scripts depend on one another, you would be better off combining them into a single file and deferring (or asynchronously executing) that script rather than applying one of these attributes to two script tags. With these attributes in play, you cannot assume the files will load in the proper order.

COMING SOON: NATIVE LAZY LOADING

Controlling which scripts block and which don't is nice, but we all know the first step in delivering top-notch performance is reducing requests, or at least orchestrating how assets are loaded. The `lazyload` attribute is a recent addition to the draft Resource Priorities spec under development at the W3C.

This tells the browser to hold off downloading the asset until all normal-priority assets have been downloaded, causing the JavaScript referenced in the `script` element to be downloaded once every other asset in the page has started downloading, freeing up the simultaneously-available TCP connections to be allocated to higher-priority stuff.

Another brainchild of that spec is the `postpone` attribute – with this, the resource is not downloaded until the element moves into the viewport area or its style is no longer set to `display:none`. Interestingly, the draft spec also supports applying this behaviour via CSS, using the `resource-priority` property:

```
main img { resource-priority: postpone; }
```

The `resource-priority` property accepts values of `normal`, `lazyload`, and `postpone`.

These proposals have me pretty excited. The speed benefit of deferring image load without having to involve JavaScript is huge, and the spec doesn't stop there: both of these attributes will be applicable to `img`, `video`, `iframe`, `object` and `embed` elements. `lazyload` will be applicable to `link` and `script` as well. You'll even be able to use these attributes in SVGs. It's almost too easy! ◘

PROFILE

Aaron (*aaron-gustafson.com*) is the author of *Adaptive Web Design*, founder and technical lead at web design agency Easy Designs, and a leading figure in the field of progressive enhancement

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/lazy-258*

ABOUT THE AUTHOR

ERIC MANN

w: *10up.com*

t: @EricMann

areas of expertise:
HTML, CSS, PHP, JavaScript, WordPress

q: what's the most heroic thing you've ever done?
a: On a backpacking expedition as a Scoutmaster, I tracked down a lost boy and his mother. They were trying to cross a canyon in the dark, with no equipment

RESOURCE

AUTHOR'S BLOG

Follow Eric Mann's blog (*eamann.com*) for daily tips on site optimisation, tutorials and discussions of new web design tools

WEB PERFORMANCE

OPTIMISE PERFORMANCE WITH LAZY LOADING

Eric Mann explains how lazy loading enables you to include rich graphical media in a website without sacrificing page performance

The continued growth of the web has led developers and content producers to cram as much into web pages as they can. High-resolution video, interactive advertisements, high-density graphics, rich visitor analytics – these can all be found on a regular web page. As consumer bandwidth and desktop capabilities increase, so does the footprint of any particular website.

At the same time, visitors are beginning to browse the web over mobile connections. This migration of site traffic from high to lower bandwidth is in direct opposition to the producers' goals of including more in their sites. New solutions are needed to combat and resolve this problem.

Lazy loading – only loading embedded assets such as high-resolution graphics and video when absolutely necessary – is one of the most straightforward solutions available to tackle this issue. It takes a little planning and finesse on the part of the development team, but can make a whole world of difference to a site's load time and overall performance.

Anything that loads outside of the main browser viewport can be loaded lazily. However, instead of optimising all the things, let's focus on two pieces of low-hanging fruit: images and video.

LOADING IMAGES

A highly engaging site is rich with graphical content. News articles and blog posts in particular are easier to consume when paired with illustrative images. Unfortunately, embedded graphics must be processed in line with the rest of the page. Higher-resolution images will load slowly and detract from the visitor experience.

Instead, we can selectively load only the images visible in the device viewport. To do so, we change our markup to replace the `src` attributes of our standard `<img />` tags with a `data-lazy` attribute. The image's source reference then points instead to a small placeholder image.

```
<img src="img/placeholder.png" data-lazy="img/horses.
hidpi.png" />
```

When the page loads, the browser will only download the placeholder image. This should be a very simple image of the fewest colours possible.

Hidden values A given website's markup might only be 20KB, but the page's images can easily total 1MB or more

The placeholder will be cached by the browser and reused, resulting in a single HTTP request to build out imagery throughout the page.

JavaScript routine

We will build a JavaScript routine into the bottom of our page to process all of the lazy-loaded images and only present the ones we want to see.

The first part of our JavaScript-based lazy loading routine is a function that can determine whether or not a given image element is in the viewport. We will include the 300 pixels immediately above and below the viewport as a loading threshold so any user scroll events will reveal already-loaded images rather than our placeholder.

> Higher-resolution images will load slowly and detract from the visitor experience

```
function inView( image ) {
  var $image = $( image ),
    view_top = $window.scrollTop() - 300,
    view_bottom = view_top + $window.height() + 600,
    height = $image.height(),
    _top = $image.offset().top,
    _bottom = _top + height;
  return height > 0 && _top <= view_bottom && _bottom
>= view_top;
}
```

If the page loads from the top, any images within 300 pixels of the bottom of the viewport will be loaded automatically. If the page loads somewhere in the middle – like when a visitor visits a URL with ▶

FOCUS ON

TECHCRUNCH

When TechCrunch began investigating a redesign in early 2013, two objectives were clear: increasing the visual engagement of the site with a deeper use of images, and making the site mobile-friendly.

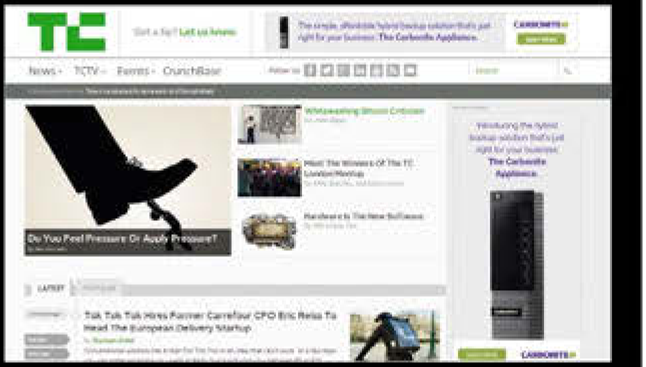
The new site design incorporates a responsive layout that is optimised both for tablet and mobile experiences. Additionally, a new editorial push requires images for every article. Feature-length articles also include a full-width, high-resolution image both on individual article pages and in the excerpt that is presented on the homepage.

It soon became apparent, however, that including more images on the page competed with the other major objective during the redesign: speed. Every one of those new article images added a significant amount of data for the browser to download. The full-width images on feature length articles were initially so large they left gaping holes in the still-rendering page.

Considering our secondary goal of enhanced page speed – the development team was in a bind. Without any optimisations, the homepage took more than 18 seconds to load in a browser with a clear cache. Our launch goal, on the other hand, was just five seconds to load from a clear cache (the goal for repeat visits was a three-second load time).

We began optimising the site by rewriting image includes to leverage the same lazy loading techniques covered by this tutorial. Further optimisations included lazy-loaded video and even on-site advertising. Social sharing icons, which can be heavy since they include scripts from third-party servers, are only loaded when the visitor actually attempts to share an article.

Lazily loading as many assets as possible helped TechCrunch achieve its targeted load times from day one.



Speeding up The new TechCrunch.com features an image-rich, responsive layout that makes clever use of lazy loading a hash – images within 300 pixels of either the top or bottom of the viewport will load.

Visible images

Next, we will define the function that actually loads visible images. This function will grab the `data-lazy` attribute and use that to replace each `<img />` tag's `src` attribute. The browser automatically detects the changes to the DOM, downloads the new assets, and replaces our placeholders in the page.

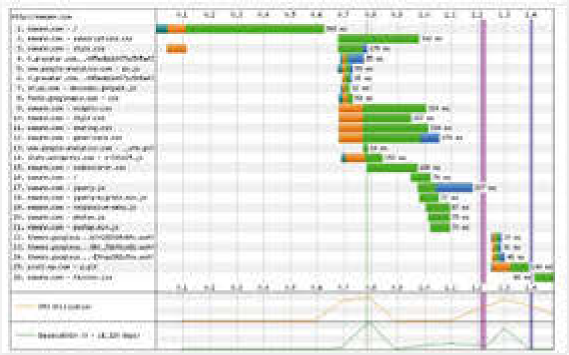
```
function maybeLoad( images ) {
  images.each( function( i, image ) {
    if ( image.hasAttribute( 'data-lazy' ) && inView( image
) ) {
      image.src = image.getAttribute( 'data-lazy' );
      image.removeAttribute( 'data-lazy' );
      image.trigger( 'lazy-load' );
    }
  } );
}
```

To improve the `maybeLoad()` function, we can automatically update the `images` collection to remove newly loaded images from the array. This reduces both the application's memory footprint and the number of elements through which we need to iterate each time.

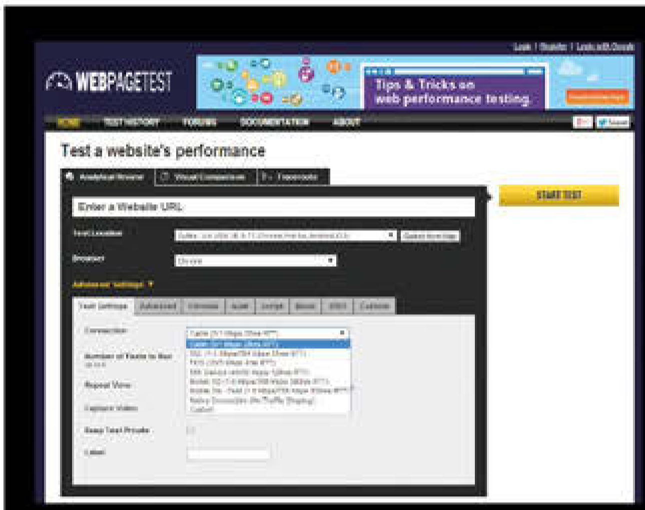
Loading threshold

As visitors scroll the page, we should also re-scan for images that have moved within our loading threshold. As the window's scroll event fires several times while the page is still scrolling, the event handler is throttled to prevent overflowing simpler devices' memories or overtaxing their processors.

```
$( window ).on( 'scroll', function() {
 if ( undefined === throttle_id ) {
  return;
 }
```



Waiting game Lazily loading assets moves the purple 'DOM Loaded' bar to the left and makes the site responsive in significantly less time



RESOURCES

TOOLS AND TRICKS

While the easiest way to test the impact of lazy loading on low-speed, low-power devices is to actually use such a device, not everyone has access to a complete device lab to do so. Instead, there are a few electronic tools you can use to grab performance data and simulate devices and connection speeds without spending a fortune on hardware.

Pingdom Network Tools

Pingdom (*tools.pingdom.com*) does more than ping sites and test them for availability. Its Website Speed Test will load your site in an emulator from any of three regions and track how long every resource takes to load.

Pingdom also provides rich analysis tools for finding any apparent bottlenecks in your site. A unified performance rating (scored out of a possible 100) is drilled down into performance recommendations.

WebPagetest

Testing from a variety of browsers in different regions over a range of connection types becomes hugely important when digging into page performance. The WebPagetest tools (*webpagetest.org*) allow you to simulate everything from the newest Chrome on a high-speed connection in Virginia, to IE8 on a 56K dial-up connection in São Paulo, Brazil.

You can see just how long it takes both new and repeat visitors to load and navigate your site. Seeing just how degraded a site can appear on a low-capability system drives home the importance of page optimisation. WebPagetest can help identify quick candidates for speeding up any site.

```
  throttle_id = window.setTimeout(
    function() {
      maybeLoad( images );
      throttle_id = undefined;
    },
    250
  );
} );
```

The event above will only fire once every 250 milliseconds, allowing us to continuously scan the page as the browser scrolls. Reducing the throttle speed will ensure even fast scrolling will never reveal an unloaded image. However, it might negatively impact slow scrolling by demanding more system resources to manage the event.

LOADING VIDEO

Video assets can also drag down a page's performance significantly. When a video is embedded in the page using a script tag, the browser processes the script tag immediately, usually loads an iFrame in its place, and then begins downloading a large Flash (or Silverlight or MP4) resource. This can lock the browser's interface and result in a visitor bouncing off your site to another. One technique to avoid this scenario is to force videos to display in a modal overlay – a lightbox – instead of in-line with the rest of your page content.

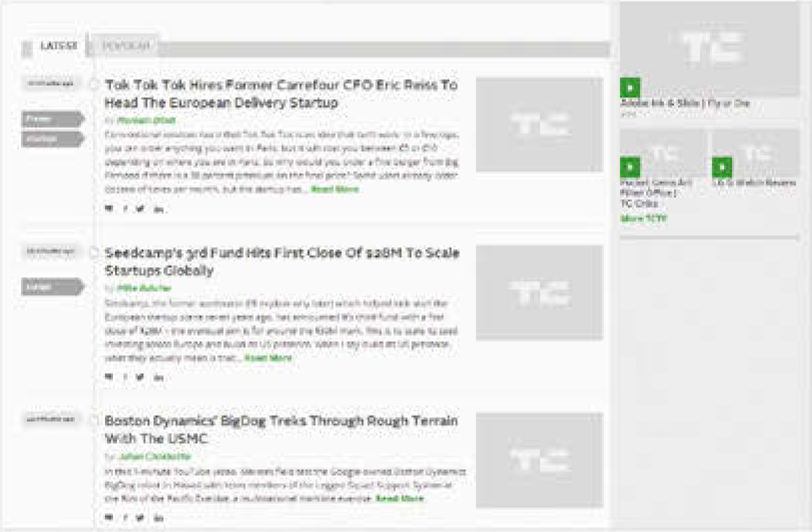
> We can set it up so the site waits for the visitor to actually click on the video before it loads

Two elements

Instead of a script embed from, say, YouTube, your page's markup will consist of two elements: a span used to present a play button, and the video's thumbnail (optimised using the lazy loading technique for images covered above):

```
<span class="iframe_play" data-embed="https://www.
youtube.com/watch?v=Xz91KWAe6t0&autoplay=1"></
span>
<img class="iframe_thumb" src="http://img.youtube.
com/vi/Xz91KWAe6t0/hqdefault.jpg" />
```

Then, we can set it up so that the site waits until the visitor actually clicks on the video before loading the video itself. We do this by listening for mouse events on the video play button (the `iframe_play` element) and invoking a custom event handler.



LATEST POPULAR

Tok Tok Tok Hires Former Carrefour CFO Eric Reiss To Head The European Delivery Startup

Seedcamp's 3rd Fund Hits First Close Of $28M To Scale Startups Globally

Boston Dynamics' BigDog Treks Through Rough Terrain With The USMC

Quick fix With the exception of the homepage feature island, all images on TechCrunch.com are lazy-loaded, including video thumbnails in the sidebar

```
$( document ).on( 'click', '.iframe_play', play_video );
```

On this user action, you have two options: Swap the placeholder thumbnail for the video player itself, or create a lightbox overlay to display the video atop the page. Our `play_video()` function will employ the lightbox technique, automatically generating and launching a modal window that plays our video.

When the lightbox closes, it's removed from the page entirely, keeping the page's markup clean and lightweight.

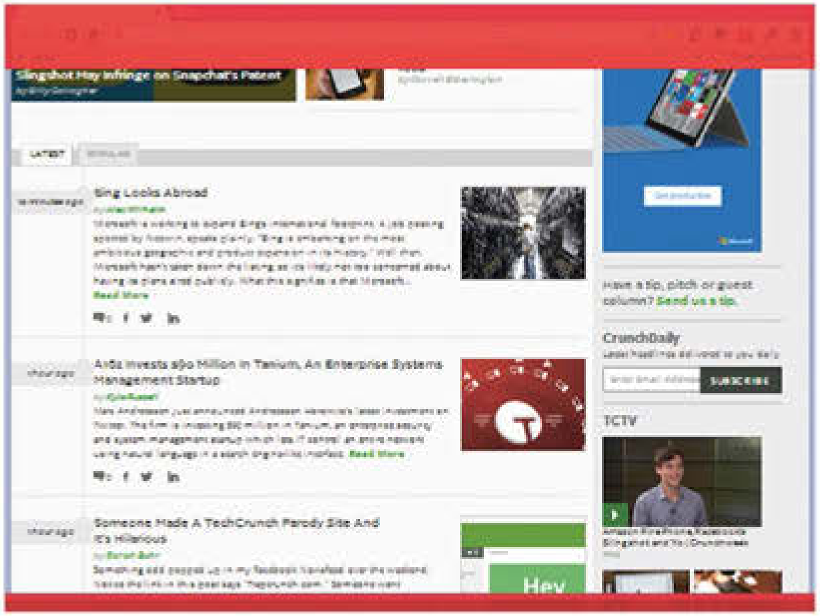
```
function close_video() {
  $modal.remove();
  $overlay.remove();
};
```

Markup

Our tool dynamically generates the markup for both a modal overlay and a container for the video. In the case of YouTube, it's using an iFrame embed to build out the video element. Since the autoplay flag is set in the URL, the video will start immediately in the modal. Clicking on the close link will remove the generated markup, immediately stopping playback when the video is removed from the DOM.

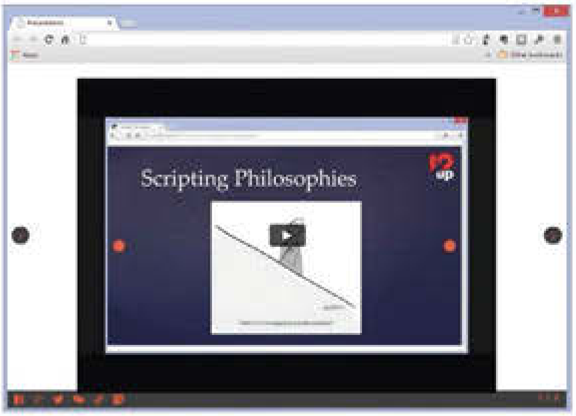
```
function play_video( e ) {
  var $this = $( this ),
    video = this.getAttribute( 'data-embed' ),
    content = document.createElement( 'iframe' ),
    overlay = document.createElement( 'div' ),
    modal = document.createElement( 'div' ),
    closer = document.createElement( 'a' ),
```

▶


Singshot May Infringe on Snapchat's Patent
LATEST
Bing Looks Abroad
Read More
A16z Invests $90 Million In Tanium, An Enterprise Systems Management Startup
Someone Made A TechCrunch Parody Site And It's Hilarious
Have a tip, pitch or guest column? Send us a tip.
CrunchDaily
SUBSCRIBE
TCTV
Hey

Right A loading threshold will force our application to load only images currently in the viewport or the red highlighted areas

Below Displaying only a thumbnail and play control on videos embedded in HTML5 presentations allows the entire presentation to load quickly



```
    $overlay = $( overlay ), $modal = $( modal ), $closer = $( closer );
    content.src = video;
    overlay.style.cssText = 'position: fixed; top: 0; left: 0; width: 100%; height: 100%; background-color: #000;z-index: 159900;';
    $overlay.on( 'click', close_video );
    modal.style.cssText = 'position: fixed; top: 30px; left: 30px; right: 30px; bottom: 30px; background-color: #fff; z-index: 160000;';
    closer.innerText = 'close';
    $closer.on( 'click', function( e ) { e.preventDefault(); close_video(); } );
    modal.appendChild( closer );
    modal.appendChild( content );
    document.body.appendChild( overlay );
    document.body.appendChild( modal );
}
```

Loading a video on-demand in a modal window provides all of the interaction of a standard embedded video. It just foregoes the overhead of loading the video's media stream when the page first displays.

FURTHER APPLICATIONS

Lazy loading web content allows a site to straddle the middle ground between providing a rich user experience and providing a speedy one. It's a clever use of engineering savvy to cut corners where possible without actually cutting down on the content displayed on the site. Thus far we've covered images and video, but just about any other content can be pulled in after the initial page load.

Sharing buttons and analytics

Sites like TechCrunch defer loading social media interactions (like Facebook, Twitter and LinkedIn buttons) until after the visitor hovers their mouse over the sharing region. With 20 articles on the homepage, this would usually mean loading 60 somewhat heavy remote includes. Waiting to load the social sharing icons until absolutely necessary is a considerable boost to site performance.

Lazy loading allows sites to balance a rich user experience and a speedy one

Other sites load analytics scripts, markup for standalone modules and even advertising asynchronously. Virtually any content that can be loaded separately from the page's main markup and assets, should be.

THE BEST OF BOTH WORLDS

Websites will continue to grow in scope and scale in order to present well-rounded experiences to those with capable hardware and seemingly-unlimited bandwidth. Presenting a similarly rich experience to those with limited access and less high-performing devices, though, doesn't need to be a daunting task. Tricks like lazily loaded media help combine the best of both worlds.

TOP TIP

GET TESTING

Make sure your testing is as thorough as your audience is diverse. Test from multiple devices, geographic locations and network types. What works for one visitor might fail for another.

ABOUT THE AUTHOR

MARK DALGLEISH

w: *markdalgleish.com*

t: @markdalgleish

areas of expertise:
JavaScript, frontend development and Node.js

HEAD TO HEAD

GRUNT VS GULP

Mark Dalgleish takes a closer look at Grunt and Gulp, two competing JavaScript build tools

GRUNT	GULP
Grunt is a Node.js-based task runner, most commonly used for frontend projects.	Gulp is a lightweight build tool, using Node's stream API to speed up and simplify complex build processes.
PHILOSOPHY	
Grunt builds are mostly made up of JSON-like configuration rather than code. Instead of writing a programmatic build script, you supply options to a series of pre-packaged build tasks inside a Gruntfile.	Gulp takes the opposite approach, favouring code over configuration. There are only a few simple methods you need to create a working Gulpfile. Due to its streaming API, builds are extremely fast.
PLUGINS	
Grunt has a large, growing community surrounding it. There are currently almost 2,300 plugins available, supporting common tasks like static asset compilation and deployment.	Gulp still has a relatively young ecosystem, but there are already around 300 plugins that have been published to npm. A core benefit of Gulp's streaming API is that plugins are small and easily connectable.
SPEED	
Tasks run synchronously by default. Each task is individually configured to interact with the file system. This I/O-heavy approach can cause performance degradation if your project grows too large. Grunt plugins try to improve the situation, but it's not ideal.	Gulp plugins usually just transform virtual files in memory, making them much faster than their Grunt counterparts. The larger and more complicated your build process becomes, the more you'll appreciate Gulp's superior performance.
SCAFFOLDING	
Yeoman, a very popular JavaScript scaffolding tool, has first-class support for Grunt. The vast majority of Yeoman generators create a project that includes a Gruntfile. There is even a Yeoman generator the can scaffold Grunt plugins.	The Yeoman team are prototyping the use of Gulp in their official generators. Yeoman was instrumental in the creation of many Grunt plugins, so this process is likely to be repeated with Gulp. Just like Grunt, there's a generator that can scaffold boilerplate Gulp plugins.

VERDICT

The Grunt team is planning to implement similar virtual file transformations to Gulp, so the choice between the two will increasingly depend on whether you prefer configuration or code for your build process. Unless your build is modified by frontend designers with little JavaScript experience, Gulp is likely to be a much more popular choice in the long run.

FACT FILE

ALTERNATIVE

Many small JavaScript projects leverage npm's 'scripts' functionality to create an extremely lean build process. This approach is particularly popular in the Node community where the more minimal solutions are often the most popular.

DEPENDENCIES

Grunt and Gulp require Node.js (*nodejs.org*).

INSTALLATION

Grunt

```
npm install -g grunt-cli
```

Gulp

```
npm install -g gulp
```

Download the files here!

Check out the full source code for this Yeoman generator at *github.com/vitkon/netmag*

ABOUT THE AUTHOR

VITALY KONDRATIEV

w: *deloittedigital.com/eu*

t: @vitkon

areas of expertise: JavaScript, frontend engineering

q: what's the worst you've ever screwed up at work?

a: an SQL delete statement where I forgot the 'WHERE' clause. It was such a relief that DevOps had an hourly backup

TOOLS

YEOMAN GENERATORS

Vitaly Kondratiev explains how to write custom Yeoman generators, which are particularly useful in multi-site and project environments

Custom Yeoman generators can be very useful in multi site and project environments and significantly improve speed of development, team collaboration and the quality of the code, while also acting as a virtual coach for more junior team members. This allows us to share best practices and ideas. In this tutorial we'll try to achieve the following with our custom Yeoman generator:

- Organise code into a maintainable folder structure. This is hardcoded into the generator and remains consistent in every project.
- Choose browser compatibility requirements (for example, Internet Explorer 8+).
- Choose a preferred preprocessor (SASS / LESS / none). Create related folders and sample style files.
- Offer a choice of UI frameworks based on the preprocessor and compatibility choices above (for example, Bootstrap 2/3, Foundation 3/4/5).
- Offer a choice of recommended libraries for common scenarios (for example, a IE8+ generator will not have jQuery 2.0 in the list).
- Choose and enforce coding conventions (check JSHint is enabled in the IDE and build process), ensuring consistent JavaScript coding conventions.

GETTING STARTED

Under the bonnet, Yeoman has three tools:

- **Grunt** (*gruntjs.com*) JavaScript task runner to help with repetitive tasks (concat, minification, Sass compilation) and other build related tasks.
- **Bower** (*github.com/bower/bower*) package manager for JS libraries.
- **Yo** (*github.com/yeoman/yo*) scaffolding tool that uses a concept of generators to create a boilerplate for a new project based on your preferences.

Start by installing Yeoman globally on your machine.

```
npm install -g yo
```

Now, you have an option to choose generators that are maintained by community to give you a quick start on the project. But what if you require a unique project and components structure, and a prompt system that will make the right components choice based on your input? That's where a custom Yeoman generator comes into play. Start by setting up globally a generator called `generator`:

```
npm install -g generator-generator
```

GENERATOR SETUP

Create a folder starting with generator-GENERATORNAME (for example, `generator-netmag`) and run the generator. Make sure the folder name starts with the `generator-` prefix and run `yo generator` in the folder.

It will ask you a few questions and create a generator core in your folder. Then you will need to create a symbolic link to your generator folder: `npm link`. This enables you to access your local generator globally, making development much easier.

Next, create a new folder elsewhere and see what you get with your newly created Yeoman generator, `yo netmag`. Not too much to look at, is it? Let's teach our netmag generator to do something useful for us.

GENERATOR LOGIC

Have a closer look at the generator's code in app/index.js. Every method you place on the `netmagGenerator.prototype` will be invoked one after another. If you need a 'private' method that doesn't have to be invoked, just put an underscore in front of the method name. For example, netmagGenerator.prototype._dontInvoke .

At the very top of the generator is our constructor function. It links up Yeoman's libraries, sets up a listener for the end event (when the methods on the prototype finish executing) and gives access to our generator's package.json file. We will leave that bit as it is for now.

We need to ask a few questions in order to make decisions on what libraries we are going to use for the new project.

For that purpose we will substitute the default askFor method with one of our own. This is what our method looks like:

```
NetmagGenerator.prototype.askForLegacy = function askForLegacy() {
  var cb = this.async();
  var prompts = [
    {
      name: 'legacySupport',
      type: 'list',
      message: 'What browsers do you intend to support?',
      choices: [
        'Modern',
        'IE8+',
        'IE7+'
      ],
      default: 0
    }
  ];

  this.prompt(prompts, function (props) {
    this.legacySupport  = props.legacySupport.toLowerCase();
    cb();

  }.bind(this));
};
```

Custom Yeoman generators can significantly improve speed of development

The prompts array contains an object for each question. In our case, we're trying to find out what browser support is required for the project.

Remember we included Yeoman's libraries at the top? This allows us to call this.prompt passing the

▶

FOCUS ON

GETTING STARTED WITH YEOMAN GENERATORS

When starting a new web project, we tend to take on too many best practices all at once. With several teams working on different projects in various locations you haven't got much time to share the knowledge and lessons learnt. Here's where your own team or company's generators come in handy. For junior developers the generator will be a guiding light in the ocean of various frameworks and solutions, acting as a virtual coach. For more senior members it will be a point where their knowledge, best practices and coding conventions can be shared and synced across the company.

A custom Yeoman generator can introduce a common project architecture and structure (such as SMACSS), define and enforce certain coding conventions, help choose required tools from a defined set of frameworks and plugins, set up testing, as well as providing some boilerplates and basic examples.

When creating a generator, having the option to react based on the response of the user creates a lot of possibilities. You can conditionally fetch dependencies, create files, tests, configuration files for various tools (EditorConfig, Git, JSHint), and anything else you can think of.

You don't need to worry much about concurrent processes, race conditions and other concerns you may have about the Node environment. It's all been taken care of. It's super easy to get started, so let's get our hands dirty with it.

If you haven't worked with Yeoman before and are looking to get up and running with it quickly, make sure you have installed Node.js, Git and, optionally, Ruby and Compass (if you plan to use Compass) and check the quick start guide at *yeoman.io*.

Yeoman generators rely on the Inquirer.js prompt system. Make sure you check *github.com/SBoudrias/Inquirer.js* if you want to dive deep into how prompt types (such as checkboxes, lists) and control helpers (for example, validation, prompt hierarchy, filtering) work.

prompts array with questions and an anonymous function for the callback that is triggered when a user gives us the answer. We then save the answer to the `this.legacySupport` variable for later analysis and invoke `this.async` to switch to the next method in our `index.js`.

The next method we're going to add is going to ask the user what CSS preprocessing is preferred, if any. This method behaves exactly like our first one and saves the answer into the `this.cssPre` variable.

Our next method is going to help the user to choose a `UIFramework` based on their previous answers. We'll dynamically build a choices array and add only those frameworks that fit the bill. For example, if a user chooses IE8+ compatibility and Sass preprocessing, we will see Twitter Bootstrap 3 and Zurb Foundation 3 in the list of options. We will again save the user's choices into the variable `this.uiFramework`.

```
NetmagGenerator.prototype.askForUiFramework = function
askForUiFramework() {
  var choices = ['None'];
  var cb = this.async();
  if (this.legacySupport === 'ie7+') {
    choices.push('Twitter Bootstrap 2');
  }
  if (this.legacySupport === 'ie8+' || this.legacySupport ===
'modern') {
    choices.push('Twitter Bootstrap 3');
  }
  if (this.legacySupport === 'ie8+' && this.cssPre ===
'sass') {
    choices.push('Zurb Foundation 3');
  }
  if (this.legacySupport === 'modern' && this.cssPre ===
'sass') {
    choices.push('Zurb Foundation 4');
    choices.push('Zurb Foundation 5');
  }
  var prompts = [
    {
      name: 'uiFramework',
      type: 'list',
      message: 'Choose your preferred UI framework',
      choices: choices,
      default: 0
    }
  ];
  this.prompt(prompts, function (props) {
    this.uiFramework  = props.uiFramework.toLowerCase();
    cb();
  }.bind(this));
};
```

Our next method will be suggesting a multiple choice of CSS and JavaScript libraries to include into our new project. And again, some options will be added dynamically based on prior answers. This is how multi-choice array looks like:

```
var choices = [
        {
          name: 'Lo-dash',
          value: 'includeLoDash',
          checked: false
        },
        {
          name: 'Modernizr',
          value: 'includeModernizr',
          checked: true
        },
        {
          name: 'RequireJS',
          value: 'includeRequireJS',
          checked: true
        },
        {
          name: 'ParsleyJS',
          value: 'includeParsleyJS',
          checked: true
        }
      ];
```

Then, to satisfy modern browsers, we will add jQuery 2.0 and we will add jQuery 1.10 to the list for legacy browser support:

```
var choiceJQuery = {
  name: 'jQuery',
  value: 'includeJQuery',
  checked: true
};
if (this.legacySupport === 'modern') {
  choiceJQuery.name += ' 2.0';
  choiceJQuery.value += '2';
} else {
  choiceJQuery.name += ' 1.10';
  choiceJQuery.value += '1';
}

choices.unshift(choiceJQuery);
```

If the user decides against using `UIFrameworks`, I suggest using Normalize.css and 960 Grid System:

```
  if (this.uiFramework === 'none') {
    choices.push(
      {
        name: 'Normalize.css',
        value: 'includeNormalize',
        checked: true
      },
      {
         name: '960 grid',
         value: 'includeGrid',
         checked: false
      }
   );
 }
```

Now that we have all the answers, let's start scaffolding our new project. The this.mkdir variable creates folders in your new folder, and this.copy copies files over from the generator folder to the new one.

```
NetmagGenerator.prototype.app = function app() {
  this.mkdir('app');
  this.copy('_package.json', 'package.json');
  this.copy('_bower.json', 'bower.json');
  this.copy('_index.html', 'app/templates/index.html');
};
NetmagGenerator.prototype.projectfiles = function
projectfiles() {
  this.copy('editorconfig', '.editorconfig');
  this.copy('jshintrc', '.jshintrc');
};
```

Don't forget to configure your .jshintrc and .editorconfig to maintain consistent coding conventions across your projects. I also prefer using JSHint (*www.jshint.com*) in conjunction with the SublimeLinter plugin in Sublime Text 2, it automatically picks up .jshintrc and shows your errors in real time as you type.

Next, we include conditions (Ruby syntax) into our _bower.json file, which contains all our dependencies. For example, what Bootstrap 3 with the LESS option enabled looks like:

```
 "dependencies": {<% if (uiFramework == 'twitter bootstrap
3') { if (cssPre == 'less') { %>
  "bootstrap": "~3.0.3"<% }} %>
 }
```

The last thing is to properly wire up all included libraries into index.html. In our generator folder we create the app/templates/ index.html, adding a bit of logic to it like so:

```
<% if (uiFramework == 'twitter bootstrap 3' && cssPre ==
'less') { %><link rel="stylesheet" href="bower_components/
bootstrap/dist/css/bootstrap.min.css"><% } %>
```

When working on large scale and long running web projects, it's crucial that we work in a unified way, in order to keep our code highly readable, consistent, well-documented, scalable and bullet-proof.

RESOURCES

FURTHER INFORMATION

Here are a few suggestions of where you can find more information about the tools mentioned in this article, including a summary of each one:

Yeoman is an OS X, Linux, and Windows friendly system-wide command-line tool that seeks out any installed "generators" on your computer and then gives them control to scaffold an app for you. Get familiar with what Yeoman can do for you out of the box and what other generators are out there that have been created by the community

When you get your head around generators and building your own custom one, you'll need to dig deeper into the prompts system. Inquirer.JS will ease the process of asking end user questions, parsing, validating answers, managing hierarchical prompts and providing error feedback in your generator.
github.com/SBoudrias/Inquirer.js

JSHint is a community-driven tool to detect errors and potential problems in JavaScript code and to enforce your team's coding conventions. It is very flexible so you can easily adjust it to your particular coding guidelines and the environment you expect your code to execute in. Don't forget to include .jshintrc in your generator to keep your code conventions and recently even code complexity requirements in sync across the team.
www.jshint.com

Bower is a package manager for your app and the bower.json file tells Bower which components to install in the app/bower_components directory. These are JavaScript and CSS components which you'll use within the website you're building. If a component is available in Bower, it's preferred to use package management rather than including component code into the generator. Bower is also capable of understanding the difference between packages needed for development (for example, unit testing tools) and those needed for development and runtime.
github.com/bower/bower

If your preferred IDE is Sublime Text, SublimeLinter is a must have plugin. It supports 'lint' programs (for example JSHint) and highlights lines of code the linter deems to contain (potential) errors in real time while you type. SublimeLinter automatically picks up your .jshintrc configuration for your project and helps to achieve great results in maintaining code conventions and consistency in your application.
github.com/SublimeLinter/SublimeLinter

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/dalek-252*

ABOUT THE AUTHOR

SEBASTIAN GOLASCH

w: *asciidisco.com*

t: @asciidisco

areas of expertise: JavaScript, open source, home automation

q: what's your favourite smell?
a: Rainbows!

DALEKJS

USE DALEKJS FOR UI TESTING

Sebastian Golasch explains how to use DalekJS for automating web application and page UI testing

Imagine for one second, you're one of the frontend developers responsible for a large online shopping website called *thames.com*. The most used feature of *thames.com*, besides shopping carts and account management, is the search box on the top of the page. If you search for an item, you'll be sent to an overview page showing all items matched by your search query. A click on the headline link of one of the items leads you to the detail information page.

So far so good, but now your boss approaches you with a new feature request. You add the code, change bits and pieces of the existing codebase and everything works perfectly, on your machine and in your browser. Quite happily you push your code, get a coffee and by the time you come back, your boss is already there, reporting that no customer is able to search for items in Chrome.

Rollback time, but one of your colleagues pushed some code in the meantime. You merge it, everything breaks ... OK, let's end this horror story here. I bet that you had similar experiences at some point in your career.

You may have tried to solve this issue by creating lists that every developer has to follow, something like this:

- Open browser X
- Type `my product` into the search box
- Click the button next to the search box
- Check if the title of the first result is `my product (new!)`
- Click the link in the headline of the first item
- Verify the headline of the detail page `my product (new!) - extended description`
- Verify the price of the item is `$2.47`
- Repeat these steps in Internet Explorer 9, 10, 11, Firefox, Safari and so on

This is time consuming and error prone. A tool to automate this workflow is required in order to be helpful, so let's consider the requirements.

HOW TO SOLVE

A tool should allow us to write this kind of automation helpers in JavaScript because that's the language all web developers have in common. Second, it should be an easy abstraction to enable frontend developers to break it down, in fact, it should be as easy as jQuery.

Speaking of jQuery, the installation should be as painless as copy and pasting jQuery plugins. Nobody likes to read tons of blog posts, experience hours of configuration and need the experience of a thousand-year-old alien time lord to get a tool working. Another important point: the tools should be able to test your page in real browsers. As awesome as headless browsers like PhantomJS are, these browsers aren't the browsers your visitors will use. They are great, but testing with them won't give you the secure feeling that your page actually works in your visitors' browsers.

Utilities we already use in our process, like GruntJS (*gruntjs.com*), should work with our tool of choice. Also, the framework should be based on a standard, if it appears to be not the tool we need, we could replace it with a tool that is based on the same standard with the lowest effort possible.

Back in the days there wasn't a single framework that matched all of the criteria! At that disappointing point in time, we decided to create our own to, finally, have a tool handy that obeys our commands.

ENTER DALEKJS

DalekJS (*dalekjs.com*) is the tool that came out of the process. It's based on the Webdriver (*w3.org/TR/2013/WD-webdriver-20130117*) specification, interacts with real browsers, has a GruntJS plugin and a really nice jQuery-like API that allows you to write your tests in JavaScript.

> Nobody likes to need the experience of a thousand-year-old alien time lord

INSTALLATION

As stated a few lines above, the installation of such a tool should be as straightforward as possible. To serve this purpose, DalekJS has only one dependency, and that's Node.js (*nodejs.org*). Bundled with Node comes npm, a package manager similar to gem or Composer. You can use it from your terminal to install the DalekJS CLI tools like this:

```
npm install -g dalek-cli
```

After this step, you should have a new command in your terminal available named `dalek`. To verify that everything is fine, just execute `dalek -v`. Then you need to navigate to the root folder of the project that you want to test and execute:

```
npm install --save-dev dalekjs
```

That will install the testing framework alongside your project. Now, the only missing step is creating a folder where your tests will end up. For the sake of brevity, we call the folder `test` and place it in the root directory of our project.

►

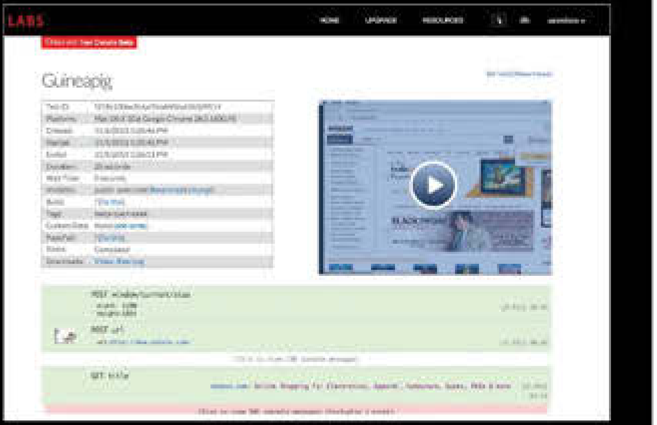
FOCUS ON

TESTING AND ROADMAPS

DalekJS has just started its evolution. If you're considering how you'll make best use of it for your projects, here are a few pointers on cloud-based testing and future DalekJS features …

CLOUD-BASED TESTING

If you don't want to run your tests on your computer, maybe because it's a Mac and you need to execute your tests in Internet Explorer, you're able to use the excellent Sauce Labs (*saucelabs.com*) cloud testing service. It offers many different browser and operating system versions and combinations to test against. Running your tests against the Sauce Cloud isn't a big deal. Just get a Sauce Labs account, install the dalek-driver-sauce plugin via npm and add it to your Dalekfile configuration as described in the plugin documentation (*dalekjs.com/docs/sauce.html*). While the plugin is in an early development stage, a few features, like the secure tunnelling, aren't yet available, but I highly encourage you to take a look at it and the possibilities of outsourcing your test runs.



Sauce Labs This cloud-based testing service offers lots of browsers and OSs

ROADMAP

If you do an emergency time shift to the near future, you'll see that DalekJS's evolution has just begun. The one feature that has the highest priority (besides fixing bugs, cleaning the internals and improving the API) is the Plugin API. With many gists and snippets lying around (useful functions for working with AngularJS applications, for example), the need to add a generic API to make them useful for the community and to build a central home for them where they can be found by other developers is very high.

Another possible feature in the near future is a Modernizr-like feature detection helper for your tests. This will become handy in progressive enhancement scenarios where your original application uses feature detection to enable and disable features for the user. Browser sniffing is the past and automated testing frameworks should respect that.

Now that the basic setup is all done, you are ready to write your first test. To keep the learning curve rising at a slow pace, let's first check if we can navigate the browser to the *thames.com* website and check if the title is as we expect it to be.

The `module.exports` variable needs to be assigned to an object that itself consists of at least one function. Within this function, you're then able to define your test logic:

```
module.exports = {
        'Title is as expected': function (test) {
                test.open('http://www.thames.com/')
                        .assert.title('Thames.
com: Online Shopping for Electronics', 'Title matches the
expectations')
                        .done();
        }
}
```

Before running this example, take a moment to explore the inner semantics of the `Title is as expected` function. As you may have figured out, `Title is as expected` is the name of our test. The only argument that gets injected into it holds all the power of defining tests. Within the Dalek world, we use the terms `actions` and `assertions` to differentiate between two kinds of methods that live in our test instance.

DEFINING ACTIONS

Actions define an interface to control the browser, pointing it to a specific URL, typing something into a form field, clicking links etc. Assertions are the counterparts of actions; they send data back from the browser and enable you to check parts of the page. Assertions are always prefixed with the `.assert` notation. In the example above, we use the `open` method to navigate to *thames.com* and then use the `title` assertion to check the content of the pages `<title>` element. Note that this isn't done via scraping the page's HTML, so if you change the page title via JavaScript, Dalek will get it. The `title` assertion takes two arguments: the first one is our value to check against, and therefore must be supplied. The second will help you to find your assertions in the test results later on.

So, the test hasn't been executed yet. Luckily, DalekJS comes bundled with the headless PhantomJS browser, so if you save the listing above in a file called `simple.js` in the newly created folder `test`, open up your terminal, navigate to the root folder of your project and type `dalek test/simple.js`. Your test will run and if everything worked, you'll get an output that looks like the console output in the tutorial files (download here: *netm.ag/dalek-252*).

RESOURCE

FURTHER READING

For more information about using DalekJS, visit the website (*dalekjs.com*), the GitHub repository (*github.com/dalekjs/dalek*) and follow DalekJS on Twitter (@dalekjs)

Checking the title of a page isn't that spectacular and you could argue that testing text contents is a waste of time, so let's dive deeper into the available functions. As indicated, our little online shop has the ability to search through its items using a Google-like search form. The form consists of two elements: a text input and a submit button.

```
<form action="search" name="site-search" accept-
charset="utf-8">
        <input type="text" id="twotabsearchtextbox"
value="">
        <input type="submit" value="Go" class="nav-
submit-input">
</form>
```

To search for a specific item, such as 'Blues Brothers video tape', you can add a second test unit to the `simple.js` file and place it right after the function you created before:

```
module.exports = {
        // ...
        'Search works': function (test) {
                test
.type('#twotabsearchtextbox', 'Blues Brothers VHS')
                        .click('.nav-submit-input')
                        .done();
        }
        // ...
}
```

This snippet will issue two actions to the browser. It will type the search term 'Blues Brother VHS' into the input field with the ID `#twotabsearchtextbox`. The other action will then look for the element with the CSS class `.nav-submit-input` and issue a click event that submits the form.

Note that typing the string into the text field will add one character after the other, just like a real user would do it. This is necessary to fire all the events that a user would also trigger if they did it by hand. For the `click` action, an invisible cursor will be placed above the element and the complete event cascade will be fired.

Assuming that submit will redirect you to some sort of search results page with matching items, you can check if the headline of the first item is as we expect. To do this, we will add two more methods to our test:

```
// ...
.click('.nav-submit-input')
.waitForElement('#result_0')
.assert.text('#result_0 .newaps a span').to.contain('The
Blues Brothers [VHS]', 'Article description is fine')
.screenshot('items.png')
.done();
// ...
```

The `waitForElement` method is needed because you need some sort of indication that the site has been loaded. Given that, in this scenario, a `#result_0` child element must exist in order to check our item, we wait for this one until we continue with other actions and assertions.

The headline describing the item in the result page is hidden in a nested `<span>` element. As Dalek supports CSS selectors, accessing the element is easy. The assertion techniques used here, checking the contents with the `to.contain` helper afterwards, differs from the assertion that we used to match the page title. This is because we're checking that the text contents of the element aren't exactly the same as requested, but at least contain the string. We call this 'assertion helper'.

To keep the boss happy, we also added a 'screenshot' method to take a picture of the current state of the items overview page.

USING REAL BROWSERS

One of the goals when creating this tool was to focus on real browsers. So far, the tests only ran in PhantomJS, which creates an experience that is quite close to Safari.

Maybe one day, Daleks will be ready to invade all web development teams out there

At the time of writing, Dalek supports Chrome, Firefox, Internet Explorer and Safari on iOS. Some of the implementations have more complete features than others, but we hope to fill these gaps in our future versions.

To run your tests in Chrome, you need to install the dalek-browser-chrome add-on. To do so, just go back to the terminal and type `npm install dalek-browser-chrome --save-dev`. Now run your tests again with `dalek test/simple.js -b chrome`. The additional parameter `-b`, the shortcut for `--browser`, tells DalekJS to run your tests in Chrome. The browser starts automatically and you can lean back to watch the magic happen.

If you want to install the Internet Explorer helper, execute `npm install dalek-browser-ie --save-dev`. If you have multiple browser helpers installed, you can chain them when starting the tests like this `dalek test/simple.js -b chrome,ie`, or you can add a `Dalekfile.json` to your applications root folder and then add the following contents:

```
{
            "browser": ["chrome", "ie"]
}
```

If you want to automate a browser in a virtual machine or on a different computer, check this screencast: *vimeo.com/78144005*.

REPORTS

Running tests this way is absolutely helpful, but after creating dozens of them, you'll get bored of scanning the terminal output manually for errors. Also, automated tests are meant to run without a human being needing to check the results. This is crucial in CI/CD environments where dozens of developers work on the same application. To tackle this specific problem, Dalek has a plugin concept for other output scenarios besides the default console output. Namely, JSON, JUnit compatible XML and HTML.

Because you're not a machine and more interested in 'human compatible' output, it's more handy to integrate the HTML reporter for demo purposes.

Adding the reporter is easy. Type `npm install dalek-reporter-html --save-dev` and execute the tests with the `-r` (short for `---reporter`) flag like so `dalek test/simple.js -r console,html`. After the test, run a new directory. `report/dalek` should appear containing an `index.html` with a list if all passed and failed tests.

CONCLUSION

Now you know the basics of how to automate your web apps tests in a sane way. There are so many more functionalities, so I encourage you to take a look at the documentation (*dalekjs.com/pages/documentation.html*), which gives hints and examples on how to use it properly.

A fair warning: DalekJS is immature. I cannot recommend basing your entire app testing on it now because it's fairly young, in developer preview mode and still needs to find its own identity. API changes and other one-time errors will happen.

However, I don't want to scare you off using it. Give it a try. If you find a bug, report it, or better, send us a pull request.

Daleks haven't gotten much love over the past few decades, but maybe one day, Daleks will be ready to invade all web development teams out there to make the web a better and less buggy place for all of us to surf in.

VIDEO

Watch an exclusive screencast of this tutorial created by the author: *netm.ag/tut3-252*

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

ABOUT THE AUTHORS

MARK MCDONNELL

w: *integralist.co.uk*

t: @integralist

areas of expertise: JavaScript, Ruby, Vim... wannabe programming polyglot

q: Which living person do you most despise?
a: Gordon's Gin

TOM MASLEN

w: *tmaslen.com*

t: @tmaslen

areas of expertise: JavaScript, HTML, CSS, doing the splits

q: Which living person do you most despise?
a: Mummy's new friend

GRUNT

8 WAYS TO IMPROVE YOUR GRUNT SET-UP

Mark McDonnell and **Tom Maslen** of BBC News talk you through eight steps to help keep your Grunt set-up fast, maintainable and scalable

Grunt (*gruntjs.com*) – which won Open Source Project of the Year at the net Awards 2014 – has quickly become an essential configuration-based command line tool within our industry for running tasks that can handle all kind of requirements. The BBC News development team use Grunt on a daily basis to make sure the *bbc.co.uk/news* codebase is tested, linted, formalised, optimised and automated.

KEEP YOUR GRUNTFILE MAINTAINABLE

One of the biggest concerns for developers working with Grunt is that this wonderfully powerful configuration file can evolve into an unwieldy monster. As with most complex tools, problems with maintainability can rapidly accumulate in a short period of time; leaving users overwhelmed with how best to resolve the complexity they're now faced with.

The best way to tackle this problem is to simplify as much as you can. If we were to take a leaf out of the object-oriented design handbook, we would know that our configuration file is doing too much and that we need to break it down into component parts to ease our ability to extend and manage our Gruntfile requirements (when we need to add more tasks and configuration settings, for example).

What we need to do is simplify our Gruntfile's structure. There are a few ways to do this, but the majority of solutions you read about boil down to different implementations of that general theme. The example I'm going to demonstrate is the best way possible to reduce the size and complexity of your Gruntfile.

In your root directory (where you have your Gruntfile) you'll create a 'grunt' folder. Inside that folder will be individual JavaScript files; each containing a different task that you would have included within your main Gruntfile.

Your directory structure could look something like the following...

```
|— Gruntfile
|— package.json
|— grunt
|     └─ contrib-requirejs.js
```

Our Gruntfile can now be as simple as:

```
module.exports = function(grunt) {
  grunt.loadTasks('grunt');
};
```

Isn't that better? It's worth noting that the string `'grunt'` that was passed to the `loadTasks` method has nothing to do with the actual Grunt object; it refers to the name of the folder you created.

You could have called the folder anything: 'omg-so-sexy', for example, in the above code would be `grunt.loadTasks('omg-so-sexy')`. With each task in its own file, we need to define the task slightly differently to how it would usually be added in the Gruntfile; `contrib-requirejs.js` should be structured thus:

```
module.exports = function(grunt) {
  grunt.config('requirejs', {
    compile: {
```

RESOURCE

PROJECT DEMO

Mark and Tom created a mini Grunt project to demo most of the techniques covered in this article. Check it out at *netm.ag/gruntdemo-256*

World service Grunt is used to manage 26 versions of BBC News client side assets. Each has separate JS & CSS; some require left to right configuration

```
      options: {
        baseUrl: './app',
        name: 'main',
        out: './app/release/main.js'
      }
    }
  });
  grunt.loadNpmTasks('grunt-contrib-requirejs');
};
```

KEEP THAT CONFIG OUTSIDE OF YOUR CONFIG!

Another important technique that we can utilise is to move specific types of configuration outside of the Gruntfile. One obvious place that we see this happen a lot is with the JSHint plug-in (*github.com/gruntjs/grunt-contrib-jshint*), which can be quite large, and so can take up a lot of space within your overall Gruntfile.

Your wonderfully powerful configuration file can evolve into an unwieldy monster

Luckily JSHint has a built-in solution to this problem. Let's take a look at how we can use it to clean up our JSHint task.

The first step that we need to take is to create a new file called .jshintrc, and within it put your JSON configuration:

```
{
  "curly": true,
  ...
}
// This is an example, you should define more than one option!
```

▶

FOCUS ON

CONDITIONALLY LOAD TASKS

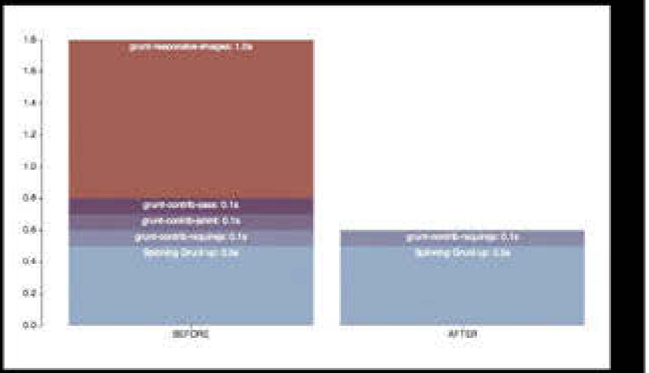
Grunt loads into memory all the tasks you add to the Gruntfile, regardless of whether or not they are going to be used. With small Grunt set-ups this isn't an issue, but as you add more tasks into your set-up it will take longer for Grunt to spin everything up before running the task you requested. This can be especially painful if you have a task that depends on something particularly heavy, such as GraphicMagick, which can take five seconds to load into memory.

So let's be a bit tricksy and set up the Grunt config in a very specific way in order to get around this problem. We can define tasks within the config whose only role is to define and run other tasks, like this:

```
module.exports = function (grunt) {
  grunt.registerTask('images', [], function () {
    grunt.config('responsive_images', {
      main: { ... }
    });
    grunt.loadNpmTasks('grunt-responsive-images');
    grunt.task.run('responsive_images');
  });
};
```

The task images is loaded into memory each time Grunt runs, but the sub-tasks within it are not. These will be loaded and run only if you run grunt images.

This will massively decrease the spin-up time Grunt needs before being ready to run a task. The only drawback is that you now have sub layers of tasks, so you'll need to give the tasks names that might describe or get confused with the tasks ran within them.



Before/after When running grunt requirejs, loading all tasks defined in my Grunt config takes 1.8s; loading only those needed for the command takes just 0.6s

IN DEPTH

RUN TASKS IN PARALLEL

A great way to speed up your Grunt running time is to run tasks in parallel. Two very popular tasks can help you do this:

- grunt-parallel
- grunt-concurrent

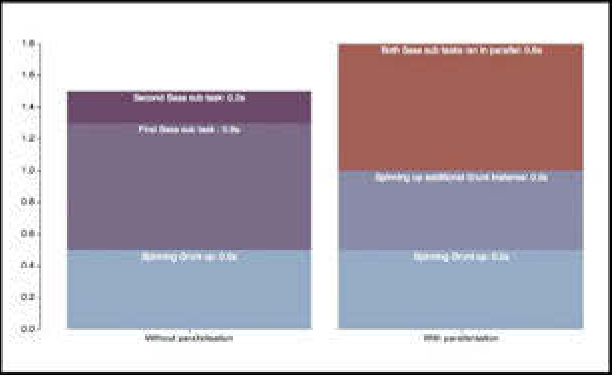
To be honest there isn't much to choose between them – we'd lean slightly towards grunt-concurrent because:

- The API is slightly more straightforward
- The project chatter on GitHub is more recent (relying on dead projects isn't fun)
- It's made by Sindre Sorhus!

Regardless of which one you choose (pick grunt-parallel if you also want to run custom – non Grunt – tasks) the one thing you should do is use it together with the time-grunt plugin (*github.com/sindresorhus/time-grunt*), which is a fantastic tool that tells you how long each task takes to run.

You've probably heard that quote before, but it's true. Before you start micro-optimising every part of your Gruntfile, the first thing you should do is measure how long the build takes to run in its current form. Then, after each refactoring, analyse the build's performance to ensure you've not introduced a regression.

For example, we recently added the grunt-concurrent plugin into our Grunt set-up; it sped up the processing of two sub tasks with RequireJS, but it actually increased the build time for our Sass tasks. This was because the two sub tasks within Sass were running at 0.8 and 0.2 seconds. Running them side-by-side with the 0.5 second penalty of spinning up a second instance of Grunt increased the time to 1.3 seconds! This is because there is a cost to running two tasks in parallel: normally about 0.5 seconds, the time it takes to spin up another instance of Grunt.



Side by side Running tasks in parallel is a great way to speed up Grunt, but measure each change you make to ensure you are reducing the Grunt compile time

Then from within your JSHint task (which we'll assume is now safely out of the Gruntfile and within its own separate task file) you can specify the location of the configuration file:

```
jshint: {
  files: ['./app/**/*.js'],
  options: {
    jshintrc: './grunt/.jshintrc'
  }
}
```

The same approach can be applied to any configuration data. It so happens that the JSHint task came pre-built with that functionality, and so with other pre-built tasks you may need to dynamically load the config file yourself using the Grunt API (see *gruntjs.com/api/grunt.file#reading-and-writing* for details).

ONLY RUN TASKS WHEN A CHANGE HAS OCCURRED

If you haven't heard of the grunt-contrib-watch task then it should be the first thing you look at next, because it's a life saver for ensuring you only run a task when the associated files with that task have actually changed.

Imagine you have a JavaScript test suite. You wouldn't want to have to manually run the task after every save of a file (especially if you're doing TDD – test-driven development), because that's a slow workflow. It would be better if you simply saved your JavaScript file … and BOOM the relevant tests are off and running! That's what this task does.

Below you'll find a simple example, which demonstrates how you can create a scripts sub-task off the main watch task, which watches all your JavaScript files – and when any of them have changed, it'll run the jshint task that has been set up separately:

```
watch: {
 scripts: {
  files: ['app/**/*.js'],
  tasks: ['jshint'],
  options: {
   spawn: false,
  },
 },
}
```

ONLY RUN TASKS AGAINST FILES THAT HAVE ACTUALLY CHANGED

The only thing that's faster than using grunt-contrib-watch to run tasks when a file changes, is to run tasks against only those files that have actually changed since the last time the task was run.

This is where the `grunt-newer` task (*github.com/tschaub/grunt-newer*) comes in handy. You define your tasks as normal, and the only thing you need to do is prefix the name of the task you want to run with `newer:`.

For example:

```
grunt.initConfig({
  jshint: {
    all: {src: 'src/**/*.js'}
  }
});
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-newer');
grunt.registerTask('lint', ['newer:jshint:all']);
```

Now when you run `grunt lint` it will only run the `jshint` task against files that have changed since the last time the `jshint` task was run.

So, if you run the task and then edit a single JavaScript file, then when that file is saved that single file will only be linted, because the `grunt-newer` task knows no other files need to be run against JSHint again.

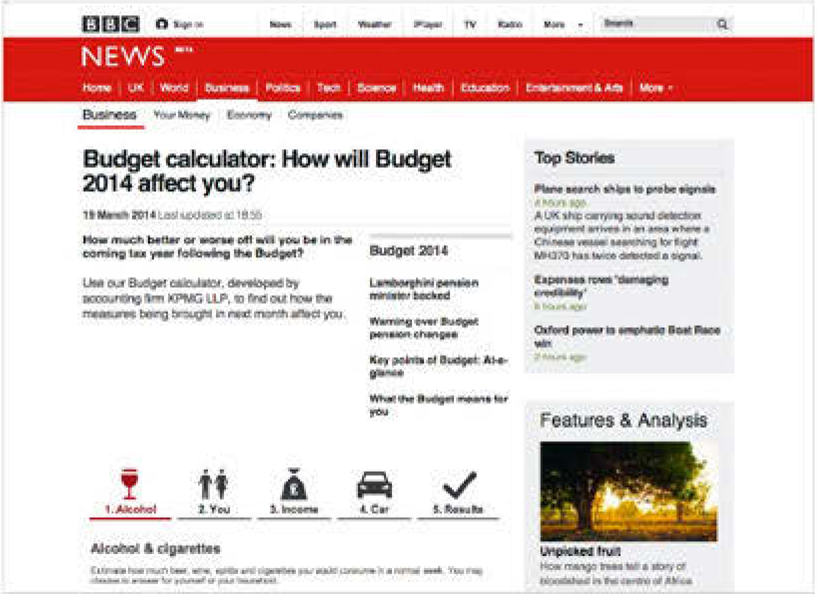
A Grunt set-up that's overloaded with tasks is obviously going to run slowly

CREATE A DEFAULT GRUNT SET-UP STARTING POINT FOR ALL PROJECTS

Grunt has a built-in feature called `grunt-init` (*gruntjs.com/project-scaffolding*). It lets you define a template project structure that gets dynamically injected with configurable values when you start a new project.

It's a command line tool configured by a JSON file. You set questions in the JSON file, these are answered on the command line, and the values are passed into the project template.

For example, imagine that you develop a large number of Node.js modules, which you publish to NPM (Node Package Manager). Rather than you having to create the same folder structure and documentation README files over and over (but only changing minor details such as the name of the library), you could create a template that `grunt-init` can utilise to set up everything automatically for you.



Budget calculator: How will Budget 2014 affect you?

19 March 2014 Last updated at 18:50

How much better or worse off will you be in the coming tax year following the Budget?

Use our Budget calculator, developed by accounting firm KPMG LLP, to find out how the measures being brought in next month affect you.

Budget 2014

Lamborghini pension minister backed

Warning over Budget pension changes

Key points of Budget: At-a-glance

What the Budget means for you

Top Stories

Plane search ships to probe signals
4 hours ago
A UK ship carrying sound detection equipment arrives in an area where a Chinese vessel searching for flight MH370 has twice detected a signal.

Expenses rows 'damaging credibility'
8 hours ago

Oxford power to emphatic Boat Race win
2 hours ago

Features & Analysis

Unpicked fruit
How mango trees tell a story of bloodshed in the centre of Africa

Alcohol & cigarettes

Estimate how much beer, wine, spirits and cigarettes you would consume in a normal week. You may choose to answer for yourself or your household.

Adds up Budget Calculator: How will Budget 2014 affect you? was a bespoke interactive created using `grunt-init`

UNDERSTAND WHAT EACH TASK DOES

The biggest criticism of Grunt is that it's slow. While Grunt does indeed contain some sub-optimal design decisions (albeit ones that are being actively addressed for the upcoming Grunt v1.0 release), a Grunt set-up overloaded with tasks is obviously going to run slowly.

By way of an example, we recently worked on a project in which we had added a 90Kb data file for D3.js (*d3js.org*; a popular data visualisation tool) to compile into an interactive map. This data file caused our Grunt build to take over two minutes to render a concatenated JS file via (`grunt-contrib-requirejs` – *github.com/gruntjs/grunt-contrib-requirejs*). It's not a great experience being forced to wait that long between saves.

The build took such a long time long because `grunt-contrib-requirejs` was creating a JavaScript sourcemap for the concatenated file, a fruitless task for a data file with thousands of points. Blacklisting the data file brought the build back down to just a few seconds.

CONCLUSION

We hope you've found this article useful. Ultimately, however, the best way to keep your Grunt set-up maintainable, fast and scalable is to understand what you are doing.

Keep reading about Grunt; follow thought leaders such as Ben Altman (@cowboy – the creator of Grunt), Sindre Sorhus (@sindresorhus – Node.js superstar) and Addy Osmani (@addyosmani – workflow enthusiast), as well as @gruntjs for the latest news on the project. The best craft people become experts in how to use their tools. ◘

VIDEO

Watch an exclusive screencast of this tutorial created by the authors at: *netm.ag/tut4-256*

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/responsive-255*

ABOUT THE AUTHOR

ANDI SMITH

w: *andismith.com*

t: @andismith

areas of expertise: Frontend development

q: When was the last time you cried?

a: The series finale to *How I Met Your Mother* was pretty emotional

RWD

LEARN TO AUTOMATE RESPONSIVE IMAGES

Andi Smith explores techniques for integrating responsive images into your site and reveals how to use Grunt to generate them automatically

Developers are obsessed with file size. When we look for a plugin, we hunt for hours for the one with the fewest kilobytes. We rewrite our entire site so we don't have to include the 'bloated' jQuery library. We minify, concatenate and uglify our JavaScript until it becomes deliberately unreadable. And then we put our tiny, tiny JavaScript file on the same page as a 400kb image.

The overall file size of a page, known as the 'page weight', is a major issue for all devices, as the success of your site may be determined by its load time, but it is of particular concern when there is a poor connection speed. If you are sending an image with a resolution higher than that of the device's screen, a proportion of the data you are sending is completely unused: if the dimensions of the image are twice that of the screen, 75 per cent of the data goes to waste. It also applies additional pressure to the device's processors, responsible for scaling the image to fit the screen. When you start adding rich animations to your page, scaling becomes especially problematic.

Responsive design techniques have solved many of the layout issues plaguing mobile users, but there are still challenges to overcome when it comes to images:

- Ensuring that the most appropriate version of an image is downloaded for the current device
- Avoiding downloading multiple versions of an image
- Determining the right time during page load to choose and load a version of an image
- Keeping markup concise
- Browser support

VIDEO

Watch an exclusive screencast of this tutorial created by the author: *netm.ag/tut5-255*

Two different techniques are now available for displaying images responsively: `srcset` and the `<picture>` element. Both are currently available in Chrome, Opera and Chrome for Android, and `srcset` is also available in Safari and iOS. Firefox has both techniques available behind a flag (*netm.ag/flag-265*). In other browsers `<picture>` can be emulated with a polyfill (*netm.ag/jehl-265*).

If you are just changing resolutions, then `srcset` should be used. For example:

```
<img src="small.jpg" srcset="medium.jpg 400w, large.jpg
800w" alt="Harry Beagle" />
```

`<picture>` uses standard media queries expressions for fine control over the displayed crop. Based on the matching media query, it uses a simplified version of `srcset` to determine the image source to load.

You can serve different images for screens with different pixel densities by adding a parameter after the filename:

```
<picture>
    <source media="(min-width: 40em)" srcset="dog-large.
    jpg 1x, dog-large-hd.jpg 2x" />
    <source srcset="dog-small.jpg 1x, dog-small-hd.jpg 2x" />
    <img src="dog-small.jpg" alt="Harry Beagle" />
</picture>
```

Additionally, we can stop our images from overflowing from their containing box by setting a `max-width` value in our CSS:

```
img {
    max-width: 100%;
}
```

If the image is not part of the page content or it is not critical to interaction but is instead part of the design, the simplest option is to include it with the `background` or `background-image` CSS property and then use media queries to load in the correct image size. We can supplement this technique with the `background-size` property to ensure our image fits the element it is contained within:

```
.container {
    background: url(/img/bg-container-small.jpg) no-repeat
    top left;
    background-size: cover;
}
@media screen and (min-width: 768px) {
    .container {
        background-image: url(/img/bg-container-large.jpg);
    }
}
```

GENERATING RESPONSIVE IMAGES

Generating a set of responsive images for your entire site manually would make it difficult to maintain. Thankfully, we can automate this process using Grunt and a task called Grunt Responsive Images (*netm.ag/grs-255*). This matches a set of files and folders and resizes the images to the dimensions you require.

Generating responsive images manually would make sites difficult to maintain

If you're unfamiliar with Grunt, it is a task runner that automates the tiresome parts of frontend development such as generating Sass files, linting JavaScript and minification. Chris Coyier has written a good article on getting started: *netm.ag/grunt-255*.

Once you have Grunt up and running in your project, adding the Grunt Responsive Images task is straightforward. Navigate to your project's directory in Terminal or Command Prompt, then type:

```
npm install grunt-responsive-images --save-dev
```

This will download the task and save it to your `packages.json` file. You also need to download an image-processing client to do the heavy ▶

FOCUS ON

THE RETINA REVOLUTION TECHNIQUE

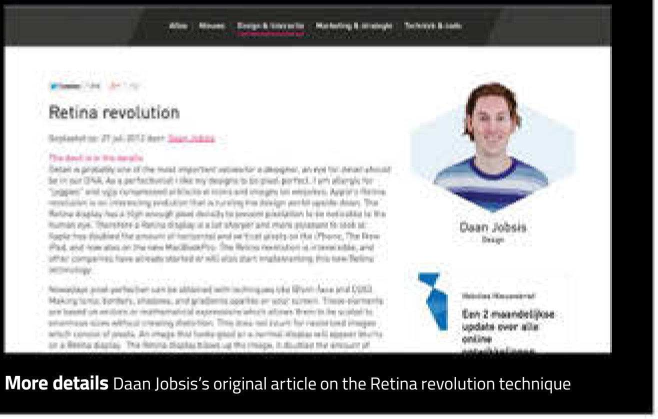
The 'Retina revolution' technique was first documented by Daan Jobsis (*netm.ag/retina-255*). Through a number of tests, he discovered that heavy compression doesn't significantly affect the quality of images with a large number of pixels, even when the file size of a high-resolution image is reduced below that of the base version. The image is then scaled within the browser, which results in a sharper image with few noticeable artifacts.

Moving forward with this discovery, one of the optimum strategies found was to save the images at 2.2 times the resolution required with a very low image quality.

To do this technique with Grunt Responsive Images, you need to enable upscaling and set a `quality` value to 25%. For example:

```
options: {
    sizes: [{ height: 440, name: 'small', quality: 25, upscale: true,
    width: 880 },
    { height: 880, name: 'medium', quality: 25, upscale:
    true, width: 1760 },
    { height: 1320, name: 'large', quality: 25, upscale: true, width:
    2200 }]
},
```

Be careful with scaling images: they require the device to run extra processing, so if you have to support older devices or you have rich animations on your site, you may experience lower frame rates.



More details Daan Jobsis's original article on the Retina revolution technique processing. Grunt Responsive Images can use either GraphicsMagick (*graphicsmagick.org*) or ImageMagick (*imagemagick.org*). GraphicsMagick is the faster of the two libraries, and the default library used. If you're using a Mac and Homebrew you can install it with:

```
brew install GraphicsMagick
```

Alternatively, install it from the official website. Once GraphicsMagick is installed, open up your `Gruntfile.js` and add the following line to load the task:

```
grunt.loadNpmTasks('grunt-responsive-images');
```

Next, add a section named `responsive_images` to the data object passed into `grunt.initConfig()` :

```
grunt.initConfig({
    ...
    responsive_images: {
    }
});
```

Within this section, we can specify multiple tasks, which is useful if we want to apply different rules to different sets of images. For example, you may wish to resize your blog's images to one size and the images in your homepage carousel to another.

Let's add a task for the homepage carousel:

```
responsive_images: {
    carousel: {
        options: {
            sizes: [{ height: 200, width: 400 },
                { height: 400, width: 800 },
                { height: 600, width: 1000 }]
        },
        files: []
    }
}
```

Automated resize Using Grunt Responsive Images, you can resize and crop your images to match your site's requirements

Our carousel task specifies three image sizes: 400x200, 800x400 and 1,000x600. By default these sizes are treated as maximum value boundaries, so the image will be resized until the first of these values has been matched. We do not have to specify both dimensions: the task can resize with just one and maintain the aspect ratio of the image.

Our task also specifies the path where we can find the source images to resize, using four parameters:

- `expand` allows us to build the files list dynamically
- `cwd` specifies the original path where we will find the files specified in `src`
- `dest` specifies the destination path to save our new images to
- `src` specifies the image file paths themselves, relative to `cwd` . Any paths we include in here will be copied to the destination structure. We can use `**/*` to specify any images that sit in this directory or any directories below the path we have specified so far; and we can use `.{gif,jpg,png}` to specifically target files with the extensions .gif, .jpg or .png.

```
files: [{
    expand: true,
    src: ['images/carousel/**/*.{jpg,gif,png}'],
    cwd: './src/',
    dest: './dest/'
}]
```

It's important the `src` and `dest` paths are different, or when the task is run a second time, it will process the images generated by the previous run.

Now run the task by going to Terminal or Command Prompt, navigating to the project directory and typing:

```
grunt responsive_images
```

The task will scan the source path and resize any images it finds, saving the output to the destination path we specified. By default, the filenames will be suffixed with `-400x200` , `-800x400` and `-1000x600` . To change this, we can add a `name` property to our size options. This has the advantage that if you change the dimensions at a later date, you do not need to change the HTML where you include these images.

```
options: {
    sizes: [{ height: 200, name: 'small', width: 400 },
        { height: 400, name: 'medium', width: 800 },
        { height: 600, name: 'large', width: 1000 }]
},
```

GraphicsMagick Image Processing System

Current Release	1.3.19 (Released December 31, 2013) download release
Development Snapshots	(Updated frequently) download development snapshots
Mercurial Repository	(Updated frequently) visit Mercurial repository

Check http://www.GraphicsMagick.org/ for the latest version of this page.

GraphicsMagick is the swiss army knife of image processing. Comprised of 282K physical lines (according to David A. Wheeler's SLOCCount) of source code in the base package (or 964K including 3rd party libraries) it provides a robust and efficient collection of tools and libraries which support reading, writing, and manipulating an image in over 88 major formats including important formats like DPX, GIF, JPEG, JPEG-2000, PNG, PDF, PNM, and TIFF.

Image processing is multi-threaded (see the multi-thread benchmark results) using OpenMP so that CPU-bound tasks scale linearly as processor cores are added. OpenMP support requires compilation with GCC 4.2 (or later), or use of any C compiler supporting at least the OpenMP 2.5 specification.

GraphicsMagick is quite portable, and compiles under almost every general purpose operating system that runs on 32-bit or 64-bit CPUs. GraphicsMagick is available for virtually any Unix or Unix-like system, including Linux. It also runs under Windows 2000 and later (Windows 2000, XP, Vista, and 7), and MacOS-X.

GraphicsMagick supports huge images and has been tested with gigapixel-size images. GraphicsMagick can create new images on the fly, making it suitable for building dynamic Web applications. GraphicsMagick may be used to resize, rotate, sharpen, color reduce, or add special effects to an image and save the result in the same or differing image format. Image processing operations are available

Power tool The GraphicsMagick library powers Grunt Responsive Images

The images generated will have kept their aspect ratio by default, and consequently will have used the width and height parameters specified as maximum value boundaries. This is useful for images within articles, but may not be as useful for a carousel where we would expect our images to have a fixed height.

Grunt Responsive Images generates responsive images for your site automatically

We can tell the Grunt Responsive Images task to resize and crop our image in order to maintain a fixed size by adding an `aspectRatio` property and setting it to `false`. We can control the crop by specifying a `gravity` value. By default, `gravity` is set to `'Center'`, but it can be set to `'North'`, `'South'`, `'East'`, `'West'`, or a combination of these, like `'NorthWest'`:

```
options: {
    sizes: [{ aspectRatio: false, gravity: 'NorthWest', height:
    200, name: 'small', width: 400 },
        { aspectRatio: false, gravity: 'Center', height: 400,
        name: 'medium', width: 800 },
        { aspectRatio: false, gravity: 'Center', height: 600,
        name: 'large', width: 1000 }]
},
```

Now we have our images correctly sized and ready for our responsive site, we can add them to our carousel using one of the responsive imaging techniques mentioned previously.

You can read more about Grunt Responsive Images on the task's GitHub page (*netm.ag/grs-255*). Many more options are available and new options are being added all the time.

RESOURCES

FURTHER INFORMATION

If you would like to know more about responsive images, the following links discuss the techniques covered in this article in more detail:

More information:

- The Grunt Responsive Images website: *andismith.com/grunt-responsive-images*
- The `srcset` specification: *dev.w3.org/html5/srcset*
- The `<picture>` specification: *picture.responsiveimages.org*
- The Responsive Images Community Group, a group dedicated to finding responsive image solutions: *responsiveimages.org*
- Ethan Marcotte's article on fluid images: *netm.ag/fluidimages-255*

More responsive image solutions:

- The adaptive images technique, a server-side responsive image technique: *adaptive-images.com*
- ImagerJS, a temporary solution for displaying responsive images created by BBC News: *github.com/BBC-News/Imager.js*
- HiSRC, a jQuery adaptive image plugin: *github.com/1Marc/hisrc*
- A up-to-date comparison chart showing the pros and cons of different responsive image techniques: *netm.ag/chart-255*

Image-processing libraries:

- GraphicsMagick: *graphicsmagick.org*
- ImageMagick: *imagemagick.org*

RICG
RESPONSIVE IMAGES COMMUNITY GROUP
JOIN US!
WHAT IS THE 'PICTURE' ELEMENT?
WHAT IS THE 'SRCSET' ATTRIBUTE?

We're a group of developers working towards a markup-based means of delivering alternate image sources based on device capabilities to prevent wasted bandwidth and optimize display for both screen and print.

Action group The Responsive Images Community Group (RICG) is a group of developers dedicated to finding markup-based responsive image solutions

ABOUT THE AUTHOR

KEVIN MANDEVILLE

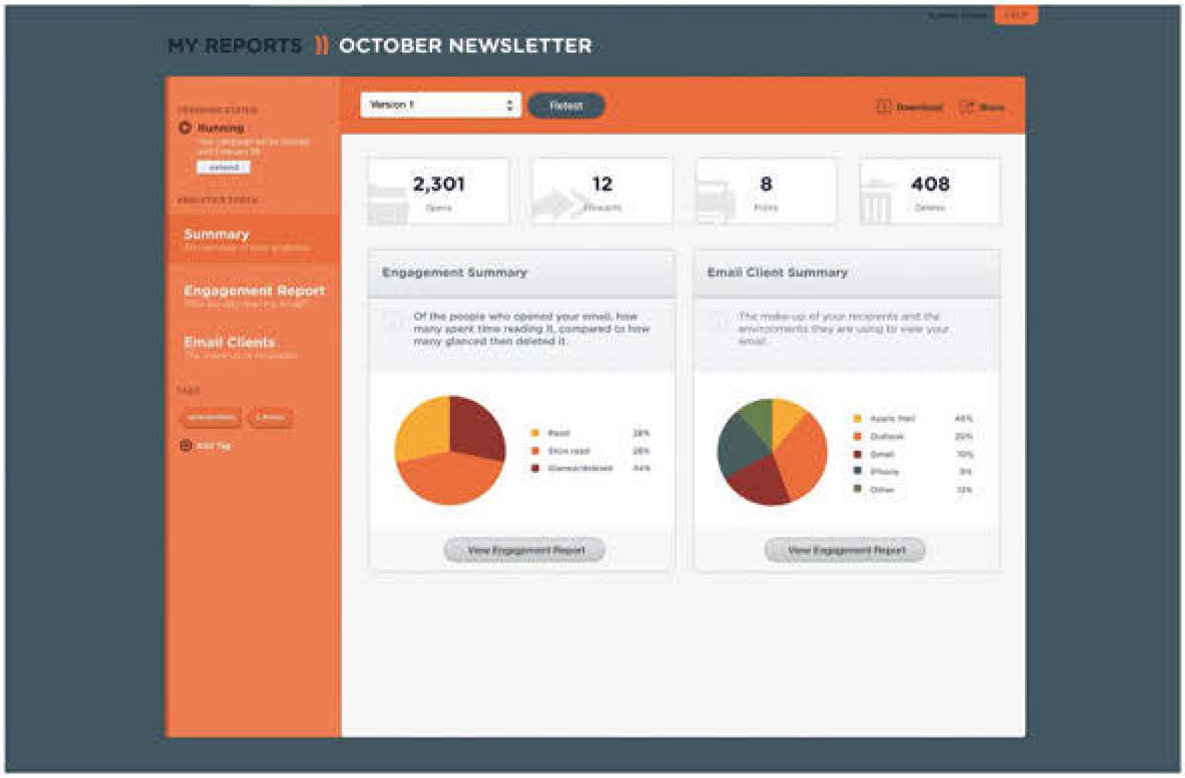
w: *litmus.com*

t: @kevingotbounce

areas of expertise: Photoshop, HTML, CSS

q: what's your opinion on Marmite?

a: No thanks. Pass the bacon



Of the people who opened your email, how many spent time reading it, compared to how many glanced then deleted it.

The make-up of your recipients and the environments they are using to view your email.

GRUNT

CREATE BETTER HTML EMAILS WITH GRUNT

Kevin Mandeville explains how to automate the process of developing HTML emails using the popular JavaScript task runner

Email clients are infamous for their poor rendering. With quirky clients like Gmail that strips the `<head>` tags out of HTML emails, or Outlook 2007–2013 that uses Microsoft Word as a rendering engine, building emails has always been problematic. Crafting an efficient workflow to develop those emails can be even more challenging.

But a lack of support for modern standards doesn't mean that email development requires anitquated tools. Emails don't need to be hand-coded in Dreamweaver like it's 1998 any more: modern tools now play a major role in the way the best email developers build their emails. In this tutorial, I'm going to focus on how Grunt can accelerate the process of preparing and testing an HTML email.

Grunt is a popular JavaScript task runner. It helps automate time-consuming tasks such as minification and inlining, and can be an even bigger time-saver for emails. If you don't have Grunt already installed, follow the instructions at *gruntjs.com/installing-grunt*.

Grunt will help us with several key functions that will automate our email development workflow:

- Removing unused CSS
- Inlining certain CSS styles
- Sending a test to the Litmus email-testing service

In this tutorial, I will go through and review each of these functions in depth.

VIDEO

Kevin Mandeville has created an exclusive screencast to go alongside this tutorial. Watch along at: *netm.ag/gruntvid-259*

Gmail decapitation Gmail strips the <head> tag out of emails – using Grunt to inline your CSS minimises potential problems

REMOVING UNUSED CSS

The first task Grunt will run for us is `grunt-uncss`. This task removes any unused CSS in the email, reducing the overall email file size. This task only works on external CSS files, not embedded styles in the `<head>`.

To install it, open your desired project location in the terminal window and run this command:

```
npm install grunt-uncss --save-dev
```

> Emails don't need to be hand-coded in Dreamweaver like it's 1998 any more

Then add the task to your Gruntfile:

```
grunt.loadNpmTasks('grunt-uncss');
```

Finally, insert the following inside the `initConfig` method of your Gruntfile:

```
uncss: {
 dist: {
  src: ['src/input.html'],
  dest: 'dest/output.css'
 }
}
```

Simply specify the CSS file you would like to uncss in the `src` array. The final output of the file is defined in the `dest` file. In the example given above, uncss will extract the CSS from `input.html` and compress it into `output.css`.

▶

FOCUS ON

USE CSS ON TABLE CELLS

Every HTML email needs to use a `<table>` structure to render properly across email clients. I recommend putting every single piece of content in its own table cell (`<td>` tags) and applying CSS to it. For instance, instead of using...

```
<h1>This is a headline</h1>
<p>This is a paragraph.</p>
<img src="example.jpg" class="mobile" />
```

... structure the content like this:

```
<table>
<tr>
<td style="/* Insert Styles Here */">This is a headline</td>
</tr>
<tr>
<td style="/* Insert Styles Here */">This is a paragraph</td>
</tr>
<tr>
<td><img src="example.jpg" class="mobile" /></td>
</tr>
</table>
```

There are a few tags that can use CSS – for example, images (`<img>`), links (`<a>`), spans (`<span>`) and client- or mobile-specific CSS. However, email clients are very quirky in rendering tags such as `<h1>` and `<p>`, so keeping all the styles on table cells in this way minimises inconsistency.

Rough plan Sketching out different email components. Each box and line of text represents a table cell that you need to style

IN-DEPTH

HOW TO TARGET OUTLOOK

Versions of Outlook for desktop (Outlook 2000–2013) are the most problematic email clients when it comes to email rendering. Luckily, Outlook can be targeted using a conditional `mso` statement.

```
<!--[if mso]>
/* Insert HTML or CSS here */
<![endif]-->
```

This hack lets us use Outlook-specific CSS styles in the `<head>` of the email or insert Outlook-specific HTML content. Additionally, specific versions of Outlook can be targeted using the following terms in the conditional statement:

- `lt` = less than a specific version
- `gt` = greater than a specific version
- `lte` = less than or equal to a specific version
- `gte` = greater than or equal to a specific version

For example, take the following statements:

- `<!--[if mso 12]>` – targets only Outlook 2007
- `<!--[if gte mso 12]>` – targets Outlook 2007 and above
- `<!--[if gt mso 12]>` – targets above Outlook 2007 (Outlook 2010/2013)
- `<!--[if lte mso 12]>` – targets Outlook 2007 and below
- `<!--[if lt mso 12]>` – targets below Outlook 2007 (Outlook 2000/2002/2003)

Here is a list of Outlook's version numbers:

- Outlook 2000 = 9
- Outlook 2002 = 10
- Outlook 2003 = 11
- Outlook 2007 = 12
- Outlook 2010 = 14
- Outlook 2013 = 15

Outlook also has its own unique CSS that can be applied to HTML elements. Here is an example of a commonly used `mso` CSS property:

```
<span style="mso-hide: all; ">This text will not display on
Outlook.</span>
```

You can find a full list of over 200 unique CSS styles by visiting *netm.ag/259-outlookcss.*

It's worth noting that in certain use cases, removing unused CSS can be a drawback when it comes to email. It is best practice to have email-client specific reset styles that aren't always referenced in the HTML, but are needed to properly render an email for some clients. Be aware of this and only use this technique where appropriate.

SETTING UP GRUNT EMAIL BUILDER

The most useful Grunt task for email is `grunt-email-builder`. It provides the best email CSS inlining functionality and the most customisable email client testing options to date. To install `grunt-email-builder`, run the following command:

```
npm install grunt-email-builder --save-dev
```

grunt-email-builder provides the most customisable email client testing options

Next, add the task to your Gruntfile:

```
grunt.loadNpmTasks('grunt-email-builder');
```

INLINING CSS

`grunt-email-builder` will help run several functions. The first is inlining our CSS. This is important because certain email clients like Gmail strip out the `<head>` tag.

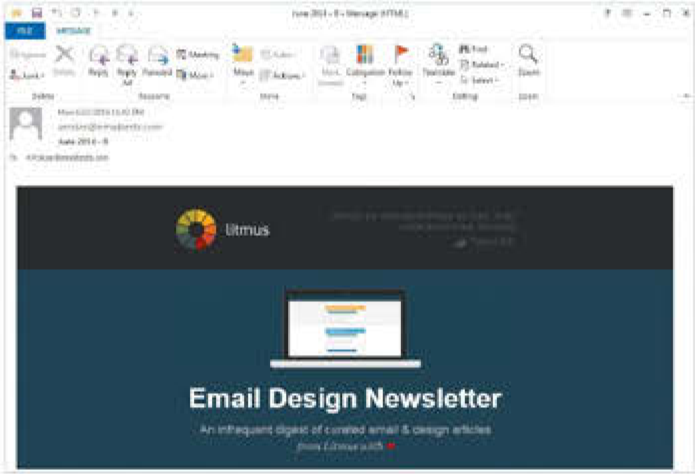
Place the following in the `initConfig` method of your Gruntfile:

```
emailBuilder:{
 inline: {
  files: { 'dest/output.html' : 'src/input.html' }
 }
}
```

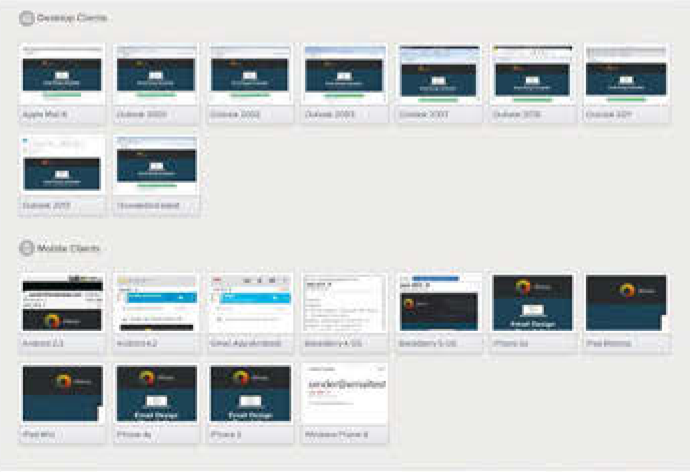
In the example above, `grunt-email-builder` will take the file `input.html` and convert it to `output.html` with inlined CSS.

`grunt-email-builder` also enables us to specify which CSS will be inlined. Using the HTML attribute `data-ignore="ignore"` on external or embedded styles prevents those styles from being inlined. Here is the HTML attribute in action:

```
<!-- External styles -->
<link rel="stylesheet" data-ignore="ignore" href="css/style.
css" />
```



Outlook issue Desktop editions of Outlook are the most problematic clients for email rendering. See the boxout for more information



Automated testing Litmus (*litmus.com*) speeds up the testing process, previewing your email in over 30 clients

```
<!-- Embedded styles -->
<style data-ignore="ignore">
/* Styles here will not be inlined */
</style>
```

For some use cases, inlining certain CSS may not have a desirable effect. For instance, an advanced email hack is to use inline styles to target Gmail and then overwrite those with new styles in the `<head>` for other email clients.

Another extremely popular CSS inlining tool for email is Premailer. The only drawback to Premailer is that it does not allow you to specify which CSS can be inlined. You can learn more about Premailer at: *github.com/premailer/premailer.*

SENDING EMAIL TESTS

It's always helpful to see a live email test in a real inbox. There are several transactional email providers that have Grunt tasks already created, such as Mandrill (*mandrill.com*) and Mailgun (*mailgun.com*). You can use your existing provider if a Grunt task already exists for it.

If you have a Litmus account (*litmus.com*), `grunt-email-builder` will also trigger an email test in Litmus to see how your email renders in over 30 email clients. This Grunt task allows users to select which email clients they want to test with. To use it, insert the following code under the `initConfig` method:

```
emailBuilder: {
 litmus: {
  files: { 'dest/output.html' : 'src/input.html' },
  options: {
   encodeSpecialChars: true,
   litmus: {
    username: 'username',
    password: 'password',
    url: 'https://yoursite.litmus.com',
    // All test options at http://yoursite.litmus.com/emails/
clients.xml
    // The <application_code> tags contain the name e.g.
Gmail Chrome:
<application_code>chromegmailnew</application_code>
    applications: ['gmailnew', 'ffgmailnew',
'chromegmailnew']
   }
  }
 }
}
```

COMBINING GRUNT TASKS

You can combine all of these functions into a single Grunt command by defining the following, under the main tasks in your Gruntfile:

```
grunt.registerTask('default', ['uncss', 'emailBuilder']);
```

The `grunt` command in Terminal will automatically trigger all of these tasks simultaneously. You can easily make changes to your files, save them and re-run all the tasks again, by simply typing `grunt` and hitting `Enter`. It's that easy.

GOING FURTHER

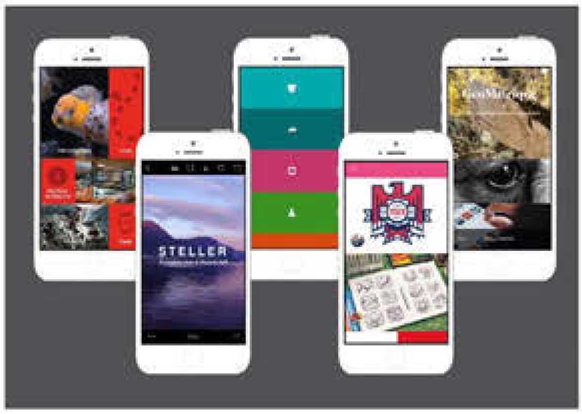
These are just a few of the ways in which you can use Grunt to automate your email development. There are plenty of other tasks you can incorporate into your own workflow, such as converting preprocessing languages such as Haml or Sass to HTML/CSS, HTML and image compression, and uploading images to a server or your ESP.

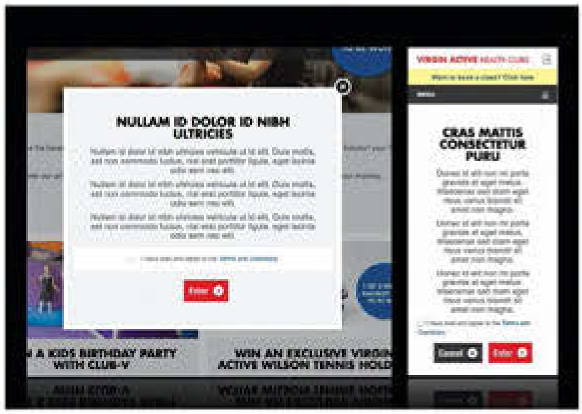
If you search for the term "grunt email" on GitHub, you will find several pages' worth of repositories showing varying email development workflows. It's all about finding the tasks that make the most sense for you to automate, and finding out which workflow suits you best.

PODCAST

Stay abreast of what's happening in the world of email design with Litmus' Email Design Podcast: *netm.ag/emailpodcast-259*

UI & RWD


STELLER


NULLAM ID DOLOR ID NIBH ULTRICIES
Enter
A KIDS BIRTHDAY PARTY WITH CLUB-V
WIN AN EXCLUSIVE VIRGIN ACTIVE WILSON TENNIS HOLD
VIRGIN ACTIVE HEALTH CLUBS
CRAS MATTIS CONSECTETUR PURU
Cancel
Enter

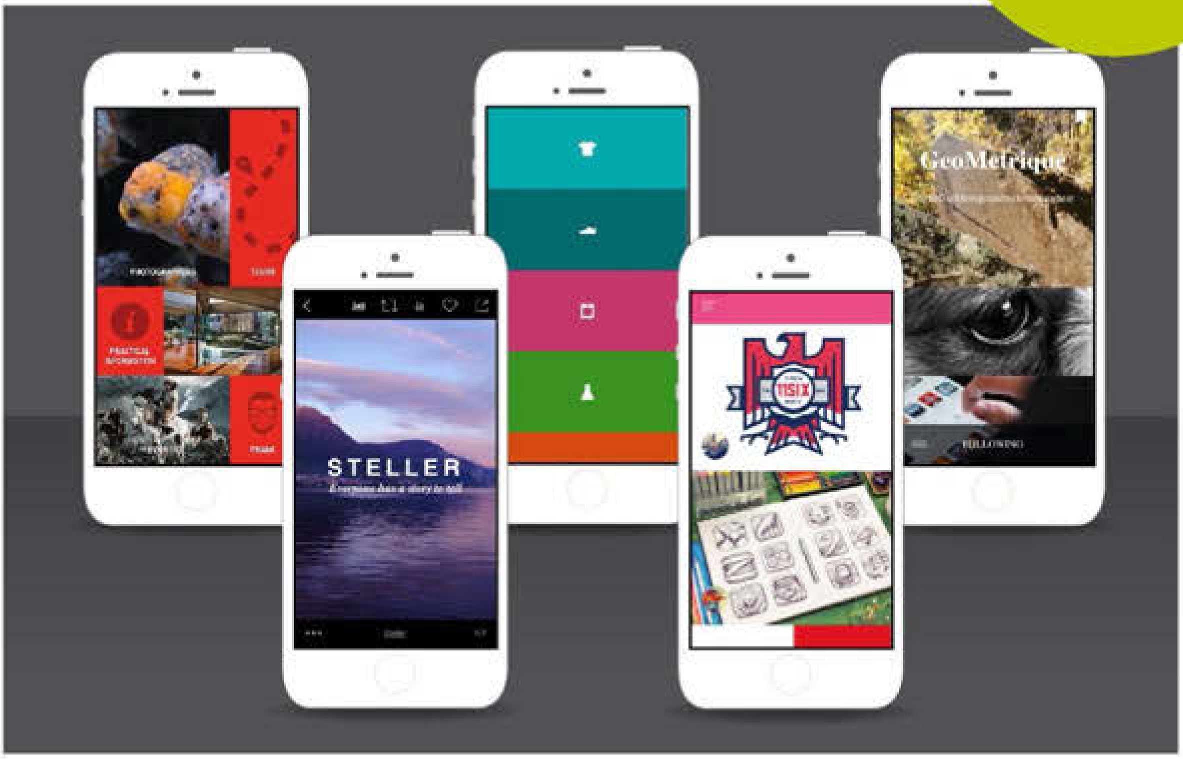
Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Download the files here!

Find the GitHub repository accompanying this tutorial at *netm.ag/famGit-260*

ABOUT THE AUTHOR
MARK JONES

w: *thisismarkup.co.uk*
t: @mark_jones
job: Lead frontend developer, Kaldor Group
areas of expertise: Frontend development, best practices and tooling
q: what was your childhood nickname?
a: Desmond



VIDEO

Mark Jones has created an exclusive screencast to accompany this tutorial. Watch along at *netm.ag/famousVid-260*

FAMO.US

BUILD SMOOTH DIGITAL UIs WITH FAMO.US

Mark Jones explains how the Famo.us framework can be used to bring native application performance to the web

60 frames per second. It's the target we should be aiming for when building digital interfaces. Despite improvements to our development tools, it is difficult – and at times seemingly impossible – to achieve animations that run smoothly and feel natural. Why is this? Browser layout engines like WebKit and Gecko were built for rendering text and links with a few images in-between, rather than the complex applications we see in browsers today.

With the success of initiatives such as Jank Free (*jankfree.org*), it's clear developers trying to create experiences matching their native counterparts are required to understand browser rendering intimately. Whilst this isn't something to discourage, Famo.us aims to provide a different option.

Famo.us is "a free, open source JavaScript framework that helps you create smooth, complex UIs for any screen". At its core is a 3D layout engine alongside a 3D physics-based animation engine. Famo.us renders to the DOM, with Canvas and WebGL rendering contexts coming in the near future.

Unlike browser rendering engines, it has been built with many similarities to 3D rendering libraries. The performance of animations has been the focus from the start of development, with the framework beginning life as code that animated a single `<div>` at 60fps without the use of CSS animations. In its current form, Famo.us' functionality focuses on building your application's views, leaving it up to you to decide how to handle other application concepts such as models, controllers and templating. At the time of writing there is no IE support due to a bug in IE's preserve-3d implementation, but support for the browser is planned once this is fixed.

To follow along with this tutorial, download the repository at *github.com/markj-/netmag-famous* and use `main.js` as your starting point. `main.js` requires the modules needed for all of the code samples, but when developing properly you will only want to require the specific modules your code uses. Each code sample has a corresponding branch in the `netmag-famous` repository that you can switch to in order to see the code in full, or if you have trouble getting an example to work.

CREATING CONTEXT

The first aspect of Famo.us to understand is the concept of a 'context'. A context is not a visible element, but instead provides the frame in which a Famo.us application will run.

```
var context = Engine.createContext();
```

The performance of animations has been the focus from the very start of development

To create something that our users will be able to see, we need to create some Renderables, then add them to our context. A Renderable is anything that can be seen on the screen in our Famo.us context, the most primitive of these being a Surface.

```
var surface = new Surface({
  content: 'My first surface'
});
context.add( surface );
```

Surfaces map to elements in the DOM tree, in this case a `<div>` containing a text node. Our content property contains a string that will be inserted into the Surface. This can be an arbitrarily long string of HTML, following the normal rules of HTML. Also available are ImageSurfaces, CanvasSurfaces, InputSurfaces and VideoSurfaces, with all of these mapping to their corresponding HTML element.

SIZE

Passing in the `size` property when we create our Surface allows us to specify width and height in a variety of ways. We can specify our Surface's ►

IN-DEPTH

UNDER THE HOOD

This tutorial covers how to use Famo.us, but now let's take a look at the optimisations that the framework uses to bring native performance to the web.

REQUESTANIMATIONFRAME

requestAnimationFrame (rAF) animations are a replacement for setInterval animations that schedule animation work to be completed during the current animation frame. The benefits include: animations only running if they're visible (saving us precious battery life) and animations being synced to your monitor's refresh rate to avoid skipped frames that would otherwise lead to janky animations.

To learn more about rAF and why we should be using it to power our animations, take a look at an excellent article by Luz Caballero at *netm.ag/request-260*.

BATCHING DOM READ/WRITES

When we modify the DOM, then query the DOM, then modify the DOM again, we run the risk of causing layout thrashing (*netm.ag/thrashing-260*) which decreases our application's performance. Famo.us' batches DOM reads and writes for us, so that all the reads happen followed by all the writes, preventing the thrashing from occurring.

FLAT DOM

Famo.us' HTML output is a flat DOM with little nesting of elements. Having a shallow DOM is one way to avoid expensive reflows (*netm.ag/reflow-260*), so despite the output being unlike something we would write by hand, it is an important optimisation to make for efficient animations.

JAVASCRIPT TRANSITIONS

There has been a trend over the past few years for developers to use CSS transitions for all animations. Frameworks and libraries such as Famo.us, VelocityJS and the GreenSock animation platform have adopted JavaScript animations instead, which allows optimisations to be made during the render loop. For more information about CSS versus JS transitions and the potential optimisations, this article by Julian Shapiro is a great read: *davidwalsh.name/css-js-animation.*

dimensions as a number (in pixels), as `undefined` (100 per cent of the Surface's parent context width/height) and also as `true` (size the Surface to the width/height of its contents).

```
var surface = new Surface({
  content: 'My first surface',
  size: [ 150, 150 ]
});
```

STYLING

Right now our Surface is looking kind of boring. We can make it more visually appealing by passing in a `properties` object during the Surface's instantiation. You'll notice these are standard CSS properties, following the rule that the HTML inside a Surface operates untouched by Famo.us.

```
var surface = new Surface({
  content: 'My first surface',
  size: [ 150, 150 ],
  properties: {
    color: 'white',
    backgroundColor: '#67b2e8'
  }
});
```

This method of styling Surfaces may seem unusual at first, but the reasoning is linked to Famo.us' rendering being abstracted from the DOM. When the Canvas and WebGL contexts are finished, we will be able to render our application to either one due to not relying on DOM-based concepts such as classes and IDs.

TRANSFORM

Our Surface will render in the top left of our context by default. To move our Surface away from the top left corner, we need to create a Modifier.

```
var modifier = new Modifier({
  transform: Transform.translate( 100, 100, 0 )
});

var surface = new Surface({
  content: 'My first surface'
});

context.add( modifier ).add( surface );
```

Modifiers let us define a collection of visual changes to apply to our Surface. The Modifier we have created applies a transform matrix, causing our Surface to be rendered 100 pixels from the top and left-hand edges of our context. Modifiers also allow us to change our Surface's opacity, size, origin and alignment.

The key to building large applications is to create smaller composable pieces

ORIGIN AND ALIGNMENT

A Modifier that changes our Surface's origin and alignment is another way that we can lay out our interface.

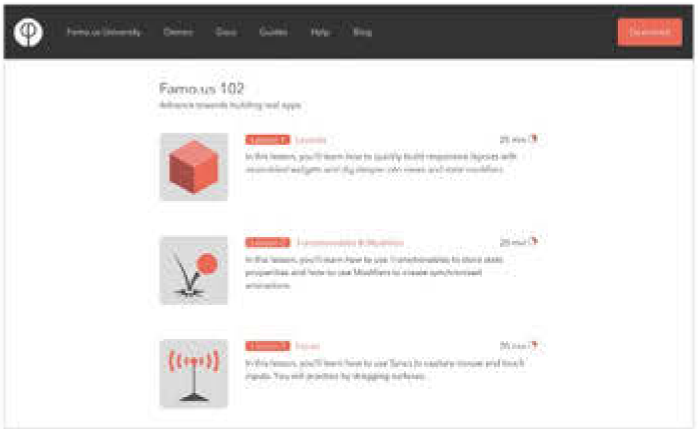
```
var modifier = new Modifier({
  align: [ 0.5, 0.5 ],
  origin: [ 0.5, 0.5 ]
});

var surface = new Surface({
  content: 'My first surface',
  size: [ 150, 150 ],
  properties: {
```

RESOURCES

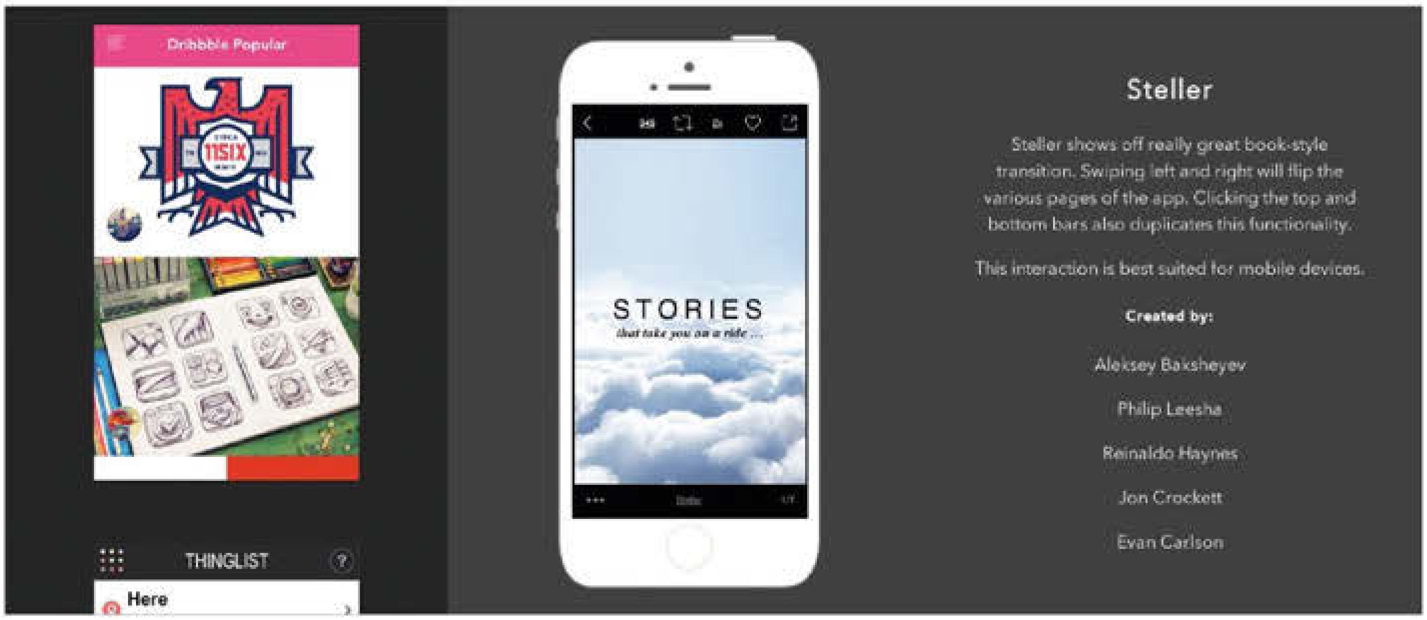
LINKS LIST

There's a comprehensive list of resources to help you find out more about developing with Famo.us at *netm.ag/hackpad-260*

Left Famo.us' stress test showcases what an early version was capable of



Right Famo.us University: your next stop on the way to mastering Famo.us



Steller

Steller shows off really great book-style transition. Swiping left and right will flip the various pages of the app. Clicking the top and bottom bars also duplicates this functionality.

This interaction is best suited for mobile devices.

Created by:

Aleksey Baksheyev

Philip Leesha

Reinaldo Haynes

Jon Crockett

Evan Carlson

Demos Play with the demos and then download the Famo.us starter kit to see the code behind them

```
    color: ‘white’,
    backgroundColor: ‘#67b2e8’
  }
});

context.add( modifier ).add( surface );
```

The align property defines the Modifier’s anchor point for the parent context, and the origin property defines the anchor point for the child Surface. These particular values will cause our Surface to be centred within our context.

Notice how the Surface adjusts its position when we change our browser window size. Try setting the align array to `[ 0, 0 ]`. This causes the centre of our Surface to anchor to the top-left of the parent context, as this is now the parent context’s anchor point.

BRANCHES AND LEAVES

To understand the effects of Modifiers we must first understand Famo.us’ render tree. Luckily, the core ideas are simple: a Famo.us context is the root of the render tree. The render tree has a Surface for each leaf. A leaf is affected by exactly the Modifiers between it and the root.

```
var surface1 = new Surface();
var surface2 = new Surface();
var modifier1 = new Modifier();
var modifier2 = new Modifier();
var modifier3 = new Modifier();
context.add( modifier1 ).add( surface1 );
context.add( modifier2 ).add( modifier3 ).add( surface2 );
```

Adding a Modifier or Surface directly to our context creates a branch. Any Surfaces added to a branch (by chaining the add method) will only be affected by the Modifiers further up the branch.

`modifier3` being added to `modifier2` demonstrates Modifier chaining, which is a pattern we can use to separate different parts of a modification. For example, if we wanted to rotate and move `surface2` , `modifier2` could handle the rotation whilst `modifier3` could handle the positioning.

VIEWS

The key to building maintainable, scalable large applications is to create smaller composable pieces that can be brought together to form our application. Famo.us views allow us to group Surfaces, encapsulate functionality and also provide an event input and output, allowing communication with other views in a decoupled manner.

Unlike other frameworks, views in Famo.us are not renderable, so they will not change our application’s appearance unless we have added Renderables to the view. Views have no default appearance; they are simply a collection of Modifiers and Surfaces, which themselves affect what is rendered. Views are treated as a single leaf on the render tree’s branch, causing any Surfaces that are added to it to be affected by Modifiers higher up the branch.

```
var surface = new Surface({
  content: 'My first Surface'
});
var surface2 = new Surface({
```

▶

FOCUS ON

FUTURE DEVELOPMENTS

Here are some of the exciting things we can look forward to in future Famo.us releases:

FAMO.US WRAPPER

The Famo.us Wrapper is similar to (and based on) Cordova (*cordova.apache.org*). The Android version of the wrapper is being built in conjunction with Intel and will allow your application to be packaged with a Chrome 35 WebView, regardless of the Android version it's running on. Not developing for Android? An optimised version of Cordova is also being worked on for iOS, Windows Phone, Kindle and Firefox.

MVC FRAMEWORK

Famo.us provides us with much of the functionality we need to build interfaces that rival native applications. However, it leaves certain architecture choices (Models, Controllers, Routing, Templating and so on) up to us. The Famo.us MVC framework will provide a complete solution for our application's architecture if you would prefer not to roll your own.

FRAMEWORK INTEGRATIONS

Already using another MVC framework? Don't let that get in the way of taking a look at Famo.us. An Angular integration is already available (*famo.us/angular*) with Ember, React and many more integrations in the works.

MOBILE TEMPLATES

If you're used to developing with Bootstrap or Ionic you'll be pleased to hear that a project is underway to port many of the templates they provide to Famo.us, allowing common application layouts to be constructed with ease. With the added bonus of 3D layout and physics, of course.

JQUERY WIDGETS

Whilst jQuery's animation techniques are starting to show their age, jQuery plugins are as popular as ever. Many of the more ubiquitous plugins are being ported to Famo.us, allowing widgets such as carousels to be swapped out for the Famo.us-powered equivalent in a snap.

```
  content: 'Another surface'
});
var view = new View();
view.add( surface );
view.add( surface2 );
context.add( view );
```

Famo.us comes with many in-built views – such as FlexibleLayout, a view for providing ratio-based responsive layouts – that provide us with a starting point for our applications. By composing several of these views, we can quickly build common application layouts with minimal effort.

CAN TOUCH THIS

Our application isn't going to be very interesting if we can't interact with it. Famo.us events are captured at three different levels. The first level is on the Surface, the second is the Surface's parent context, and the third and final level is the Engine.

```
surface.on( 'click', function( e ) {
  console.log( 'Surface clicked', e );
});

context.on( 'resize', function() {
   console.log( 'Context resized' );
});

Engine.on( 'keyup', function( e ) {
  console.log( 'Keyup event', e );
});
```

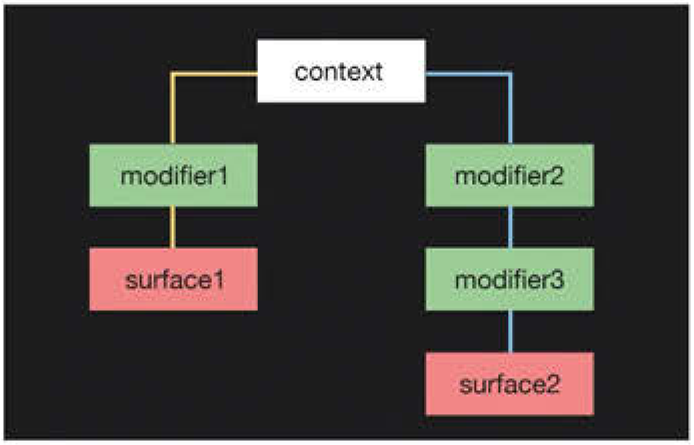
ANIMATIONS

We've covered many of Famo.us' basics, but we still seem to be focusing on displaying content, something that modern browser engines are pretty good at. Be it the animation of a scrollview or the transitions between screens and menus, our animations have to be smooth and feel natural to the user's input.

Famo.us achieves this by doing away with CSS animations, instead applying matrix transformations directly to the Surfaces it is animating. We will use a specific type of Modifier called a `StateModifier`, which is simply a Modifier which contains its own state via `setTransform`.

```
var context = Engine.createContext();

var surface = new Surface({
  content: 'My first surface'
});

var modifier = new StateModifier();
```

Build highly interactive, data driven, and modular web applications with the combined strengths of Famo.us and AngularJS.



Left The Angular integration is ready and waiting for developers with more integrations on the way

Right Say goodbye to the DOM and hello to the render tree

```
context.add( modifier ).add( surface );

modifier.setTransform(
  Transform.translate( 100, 300, 0 ),
  { duration : 1000, curve: Easing.inOutBack }
);
```

`StateModifier`'s `setTransform` method allows animations to be triggered on Surfaces. The first parameter determines what we want to animate (such as rotations or opacity), and the second describes how it should perform the animation, including the duration and the type of animation curve. `setTransform` has a third parameter – a callback that runs once our animation is complete.

Where Famo.us excels is the combination of its physics engine with animations

We're using Famo.us' Easing object to dictate the curve of our animation. The Easing object provides us with over 30 easing curves to use in our animations. Should we wish to sequence several animations, we can chain them by calling `setTransform` multiple times. When we chain animations, the previous animation will finish before the next one begins. Complex sequences of animations are simple to create using this pattern.

PHYSICS

Where Famo.us really excels is the combination of its physics engine with animations. Our previous animation occurred over a fixed amount of time. Famo.us gives us the ability to create animations that occur for varying durations, depending on the way an element is interacted with, and with a choice of transition styles.

```
Transitionable.registerMethod( 'wall', WallTransition );

var surface = new Surface({
  size: [ 150, 150 ],
  properties: {
    backgroundColor: '#67b2e8'
  }
});

var modifier = new StateModifier();

context.add( modifier ).add( surface );

var wall = {
  method: 'wall',
  period: 500,
  dampingRatio: 0.1
};

modifier.setTransform(
  Transform.translate( 0, 400, 0 ),  wall
);
```

This example shows how to register a Transitionable method called `'wall'`, which we use as the second argument to our `StateModifier`'s `setTransform` method. This causes our Surface to animate 400 pixels from the top of our context over a duration of 500 milliseconds, and when it arrives at its destination it bounces as if it has hit a wall, causing it to come to a gradual stop.

WHAT'S BELOW THE SURFACE?

Famo.us is an exciting framework that is simple to use whilst being incredibly powerful. This article couldn't cover everything it does, so check out the resources in the sidebars to find out more.

REVIEW

DAVID FETTERMAN

With thanks to David Fetterman, head of Famo.us Labs, for his review of this tutorial.

Download the files here!

Download all the files you need for this tutorial at *netm.ag/UIs-261*

ABOUT THE AUTHOR

DAN NAGLE

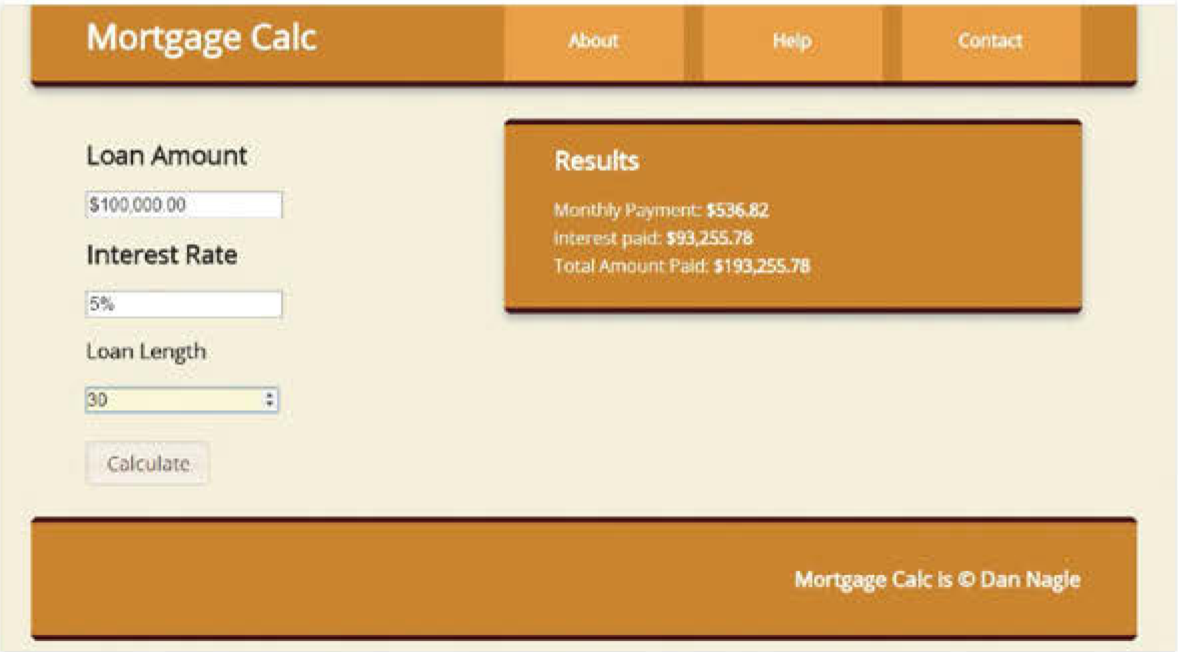
w: *DanNagle.com*

t: @NagleCode

areas of expertise: Desktop and web apps, Linux, HTML5 game development

q: who would play you in the movie of your life?

a: The Rock



Mortgage Calc

Loan Amount

$100,000.00

Interest Rate

5%

Loan Length

30

Results

Monthly Payment: $536.82
Interest paid: $93,255.78
Total Amount Paid: $193,255.78

Mortgage Calc is © Dan Nagle

USER INTERFACES

LAY THE GROUNDWORK FOR A MODERN WEB UI

Dan Nagle shows you how to leverage H5BP, jQuery and Mustache.js to build the beginning of a web user interface

I find the easiest way to learn a new tool or technique is with a project. Without a concrete end goal to motivate me, the lesson is merely a case of absorbing disjointed facts – and I find facts without context nearly impossible to remember. The goal of this tutorial is to lay the groundwork for a modern web interface, and as an example project, we'll be building a mortgage calculator. This is easy to understand with only a few variables to juggle, and the calculation can be quickly checked using another online calculator.

To build my web UI, I will be using a number of different tools. HTML5 Boilerplate will supply a solid foundation to start the build process. I will then add jQuery so the UI structure has a consistent and pleasant theme. A template engine – Mustache.js – will be used to begin the actual app.

However, this tutorial will only cover the bare-bones calculation. After all, the goal is to lay the groundwork for further development.

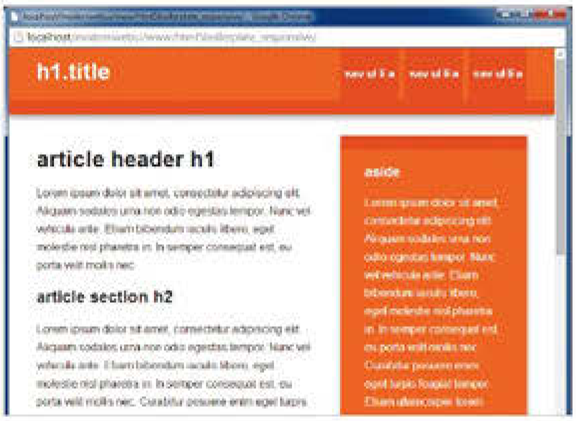
HTML5 BOILERPLATE

First off, create a file called `index.html` and begin writing `<!DOCTYPE html><html>`. No, that is silly. We are going to take a massive shortcut and download HTML5 Boilerplate from *initializr.com*. HTML5 Boilerplate (now referred to as H5BP) is a template that encompasses dozens of best practices when developing a modern website. The high points of the included package are:

- A very thorough index.html
- Basic JavaScript libraries, such as jQuery and Modernizr

VIDEO

See the tutorial in action in Dan Nagle's exclusive accompanying video at *netm.ag/UIsVid-261*



h1.title

article header h1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam sodales urna non odio egestas tempor. Nunc vel vehicula ante. Etiam bibendum iaculis libero, eget molestie nisl pharetra in. In semper consequat est, eu porta velit mollis nec.

article section h2

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam sodales urna non odio egestas tempor. Nunc vel vehicula ante. Etiam bibendum iaculis libero, eget molestie nisl pharetra in. In semper consequat est, eu porta velit mollis nec. Curabitur posuere enim eget turpis

aside

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam sodales urna non odio egestas tempor. Nunc vel vehicula ante. Etiam bibendum iaculis libero, eget molestie nisl pharetra in. In semper consequat est, eu porta velit mollis nec. Curabitur posuere enim eget turpis feugiat tempor. Etiam ullamcorper lorem

Templates 'Responsive' is a dynamic layout without a UI engine

- Basic CSS libraries, such as Normalize.css
- Asset libraries, including a 404 page, .htaccess, favicon and touch icons

Essentially, H5BP gives you all the essentials, so instead of building each element from scratch, it's just a case of modifying things.

By downloading H5BP from Initializr, we are also given a choice of themes. We want a free layout to go with our H5BP. Layouts on offer include 'None' (the default template), 'Bootstrap' (which has a corporate feel) and 'Responsive', which we will be using. As you might expect, with this layout the elements are rearranged to suit whatever screen they are being viewed on. The Bootstrap theme has responsive elements, too.

H5BP will supply a solid foundation, and jQuery will be added so the UI has a consistent theme

The technique specifies a different layout for different screen sizes. This is done through media queries, a technique introduced with CSS3.

It works as follows:

```
@media only screen and (min-width: 480px) { /* css */ }
@media only screen and (min-width: 768px) { /* css */ }
```

When the screen reaches 480px or higher, the additional CSS becomes active. When the screen reaches 768px, the next layout becomes active. This allows the webpage to adjust itself. Before CSS3, the original technique was to parse browser user agent strings to determine if a device was a mobile, and serve a different page. It was unreliable.

FOCUS ON

DYNAMIC DIALOG BOX

jQuery UI (*jqueryui.com*) is a companion framework to jQuery that is also developed by the jQuery Foundation. It includes a lot of handy widgets, such as Dialog, Slider, Datepicker and Tabs.

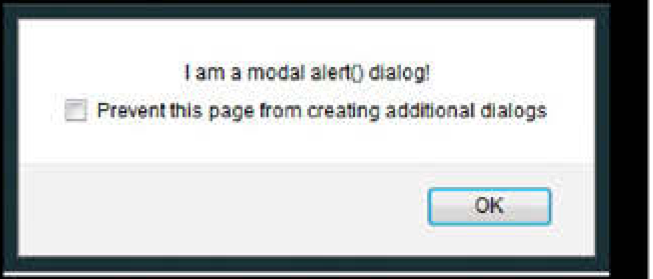
One very common task is to toss up a dialog when the user clicks a button or link. Often, the default browser alert box is used. This is fine until it is launched twice in a short amount of time – then the browser will ask the user to suppress it.

To simultaneously prevent alert suppression and make the dialog more pleasant, use a jQuery UI dialog. The `alert()` is convenient in that it can be added on-the-fly and unreliant to the DOM. A jQuery UI dialog can do this too, but it requires more thought. The example below shows how to do it:

```
$('<div/>', { text: 'Hello Dynamic Dialog.'
 }).dialog({ title: 'Sample Dialog Title.',
 width: 300, height: 200, modal:true,
 close: function(){ $(this).remove();  },
 buttons: {
   "OK": function() {$(this).dialog("close");  }
} });
```

First we create a blank `div` with the text `Hello Dynamic Dialog`. The text inside the `div` will be the content of the dialog. `modal` forces the user to acknowledge or dismiss the dialog before using the main screen. `close` is binded to a remove function. We must tell the dialog box to remove itself or it will be attached and orphaned in the DOM, causing a memory leak.

It may not be a pure unrecoverable leak like in C or C++ programming, but it's still worth tackling. It's allocating an object (a `div`), hiding it, and never using it again. If that happens too often, it will bog down the browser and make your site feel slow.



I am a modal alert() dialog!

□ Prevent this page from creating additional dialogs

Shhhh Alert suppression helps users silence noisy sites – it can also break your UI

IN-DEPTH

UGLIFYJS

Before handing your project over to an end user, you should minify and consolidate your code. This is very important for two reasons:

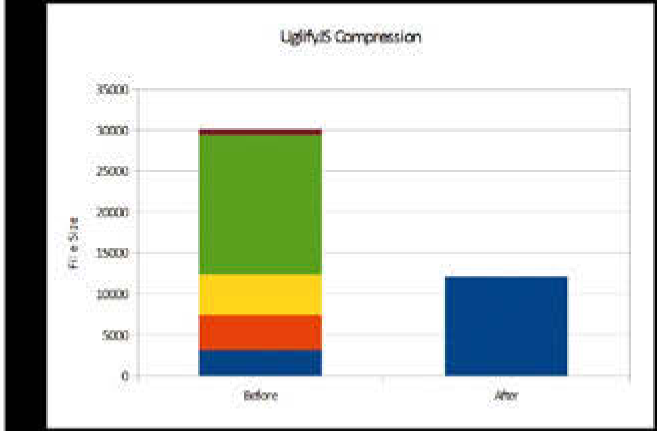
1. Your site will be faster because the browser has fewer files to fetch
2. Your site will be harder to hack or steal because your code will be harder to read

UglifyJS (*github.com/mishoo/UglifyJS*) is designed to do just this. The only downside to using it is that troubleshooting on the production server is more difficult. For the same reason hackers will have trouble reading your code, you will have trouble debugging it. UglifyJS has numerous parameters to help automate code compression (via Grunt) and really squeeze as many bytes out of your files as you can.

Assuming all your `.js` files are in a single directory, the absolute simplest way to get started is to minify the files in that directory with these commands (for Windows):

```
del main-ugly.js
del app-joined.js
TYPE *.js >> app-joined.js
uglifyjs app-joined.js > main-ugly.js
```

Now, replace that series of `.js` files with `main-ugly.js` in your web app.



Bandwidth savings UglifyJS Compression showing 60 per cent bandwidth savings for just one directory

JQUERY UI

The Responsive theme we have chosen from Initializr does not have a UI framework, so we must provide one. jQuery UI is a framework of CSS classes, sliders, dialogs, icons and so on. It is a solid choice for this task. Bootstrap also has a nice UI framework, which we could easily have chosen instead of jQuery UI. Indeed, Bootstrap is the one of the most starred projects on GitHub. Both frameworks have specific strengths and weaknesses, and the choice is mostly down to personal preference. Right now, Bootstrap probably has a slight edge over jQuery UI in terms of support and development speed. Some sites combine the two libraries and leverage both.

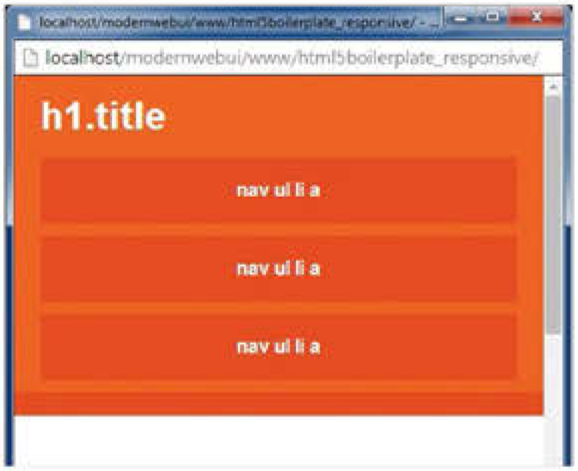
The first task is to pick a theme from *jqueryui.com/themeroller* or roll your own. I highly recommend choosing an existing theme and using another CSS file for overrides. For our mortgage calculator, I chose the theme 'Humanity' – we want to use an existing theme so we can use Google's servers to host it. See the code below for how to do this:

```
<link rel="stylesheet"  href="//ajax.googleapis.com/ajax/
libs/jqueryui/1.11.0/themes/humanity/jquery-ui.css" />
```

Every standard jQuery UI theme in the gallery is available on Google's public hosted libraries (*netm.ag/libraries-261*) – that's over 20 themes to experiment with. The example given by Google uses 'Smoothness'. Simply replace Smoothness with the name of any theme from the jQuery ThemeRoller, and it will load.

The `//` at the front of the link makes fetching the library protocol agnostic. If your browser is using

Web fonts There are a wealth of web fonts available from Google - as shown by this free-font pairing project from @femmebot (*netm.ag/fonts-261*)



localhost/modernwebui/www/html5boilerplate_responsive/

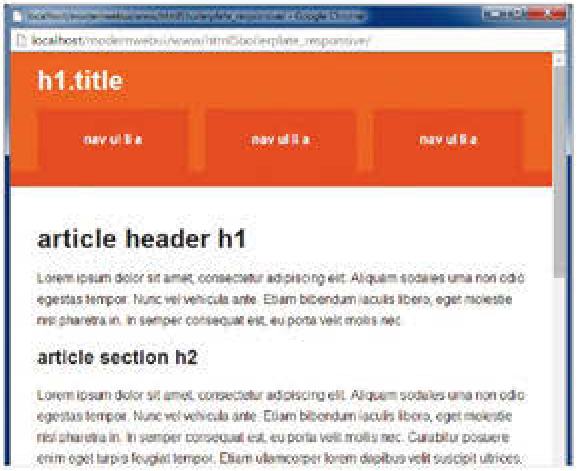
h1.title

nav ul li a

nav ul li a

nav ul li a

Phone view In 'Responsive', the elements adapt depending on screen size



localhost/modernwebui/www/html5boilerplate_responsive/

h1.title

nav ul li a

nav ul li a

nav ul li a

article header h1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam sodales urna non odio egestas tempor. Nunc vel vehicula ante. Etiam bibendum iaculis libero, eget molestie nisi pharetra in. In semper consequat est, eu porta velit mollis nec.

article section h2

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam sodales urna non odio egestas tempor. Nunc vel vehicula ante. Etiam bibendum iaculis libero, eget molestie nisi pharetra in. In semper consequat est, eu porta velit mollis nec. Curabitur posuere enim eget turpis feugiat tempor. Etiam ullamcorper lorem dapibus velit suscipit ultrices.

Tablet view The tablet layout kicks in when the screen is over 480px

`http` or `https`, then that will be used. Note that this method will not work if viewing the html locally using the `file://` protocol.

While you are adding the remote theme, modify your H5BP template to use Google's public jQuery library for both regular jQuery and jQuery UI:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
<script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.11.0/jquery-ui.min.js"></script>
```

There's one last stop before leaving Google's public hosting. If you are not pleased with the default font options, Google provides lots of great free fonts that you can choose from. I recommend Open Sans – a nice clean font with high readability, or Play, which has more of a techy feel.

Loading a custom font creates a performance hit. Play around, but only link to the ones you need. If you are going to rely on a third-party server to host some of your site's files, make sure the CDN is widely used and trusted.

REMAINING PLUGINS AND LIBRARIES

There are just a couple of remaining libraries to toss into our setup. These are as follows:

- jQuery formatCurrency (*netm.ag/format-261*) is a plugin that allows easy currency manipulation (currency symbol, thousands separators, and two decimal places)
- Mustache.js is a JavaScript implementation of the minimalist Mustache templating engine. Although it is very lightweight (it supports no logic statements), it is more than suitable for our needs – but any website of real complexity should invest in templates for a consistent appearance

This concludes all our helper libraries. Now it's time to piece everything together and write some code.

VERSIONING

Before beginning on the JavaScript code, rename `index.html` to `index.php` (or your language of choice). Yes, all our business logic is in the JavaScript, but the server will provide a very important service for us. Have you ever made a change in a CSS or JavaScript file and not seen the result because of aggressive caching? This happens frequently.

There are numerous tricks to try to fix this, but most are unreliable – browsers will cache anyway, particularly on mobile devices where bandwidth is very expensive. However, there is one very easy trick that does work, and it is to version the files. We will use our web server to do this automatically. See below for the PHP version:

> **Mustache.js is a lightweight JavaScript version of the Mustache templating engine**

```
<link rel="stylesheet" href="css/app.css?v=<?php echo filemtime("css/app.css");?>">
```

The end result is a UNIX timestamp of the file modification date appended to our link. The result looks like this: `css/app.css?v=1407624474`.

Whenever we save the file, the modification date changes, and that time stamp will change. Our web browser will think it is a new file and re-fetch – cache problem solved. You'll need to do this for ▶

every local asset that may frequently change (such as style sheets and JavaScript files).

Note that `filemtime()` returns a 32-bit signed integer. This integer timestamp will turn negative and drop to the year 1901 in the year 2038. This has no effect on our caching system – we only care about the time changing to force a browser reload. However, we're creating a mortgage calculator, and many new mortgages will last past 2038. JavaScript dates are not susceptible to this particular problem, but many of the underlying Linux apps and services are. If you are using PHP, use the DateTime class (version 5.2+) to manage dates.

ADDING THE MORTGAGE LOGIC

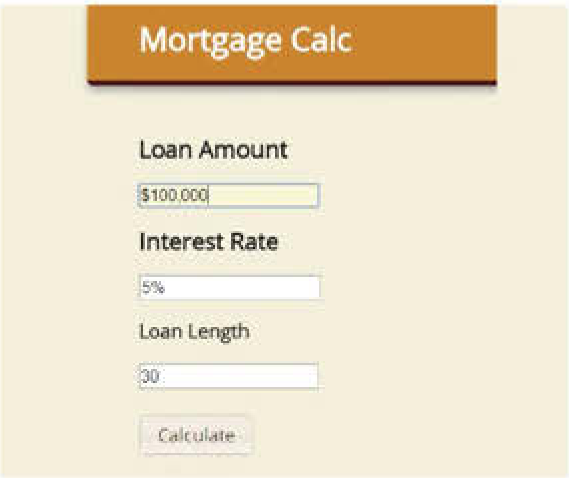
I am going to skip the details of the maths behind the actual mortgage calculation. Instead, I will ask you to consider the following two monthly calculations. One is PHP, the other is JavaScript. The exponentiation function, `pow()`, that appears in both calculations, takes a variable and raises it to the power of another variable.

For PHP:

```
    $monthlyinterest = floatval($interestrate / (12 * 100));
    $denominator = 1  - pow( 1 + $monthlyinterest,
0-$nummonths);
    $monthlypayment = floatval($originalloanamount) *
floatval($monthlyinterest / $denominator);
```

For JavaScript:

```
     var monthlyinterest = (interestrate / (12 * 100));
    var denominator = 1  - Math.pow( 1 + monthlyinterest,
0-nummonths);
    var monthlypayment = (originalloanamount) *
(monthlyinterest / denominator);
```


Mortgage Calc

Loan Amount

$100,000

Interest Rate

5%

Loan Length

30

Input ui-state-highlight as applied to the focus() event of an input box

RESOURCE

SLIDEDECK

Take a look at the slidedeck that goes alongside this tutorial at *naglecode.com/modernwebui*

The two functions perform the same task, but with one very important difference: PHP executes on the server while the JavaScript executes in the browser. With PHP, every user wanting to calculate a mortgage will ask your server to perform the very expensive `pow()` calculation and wait for a result. I can assure you from real world experience that this will kill a cheap shared hosting plan once a few hundred users start accessing it simultaneously.

Whenever possible, you should strive to put logic in the web browser to ease the load of your server. The idea is very simple: instead of having one CPU doing 1,000 calculations (the server), you will now have 1,000 CPUs doing one calculation (the user's web browser). This very reason is also why we are using JavaScript Mustache.js templates instead of PHP Smarty templates.

Whenever possible, strive to put logic in the web browser to ease the load of your server

ADDING THE UI LOGIC

Our `mtgCalc()` function only accepts three fields, so the form we need to place on our web page is very simple. There are a couple of interesting things we can do beyond `<input type=text />`.

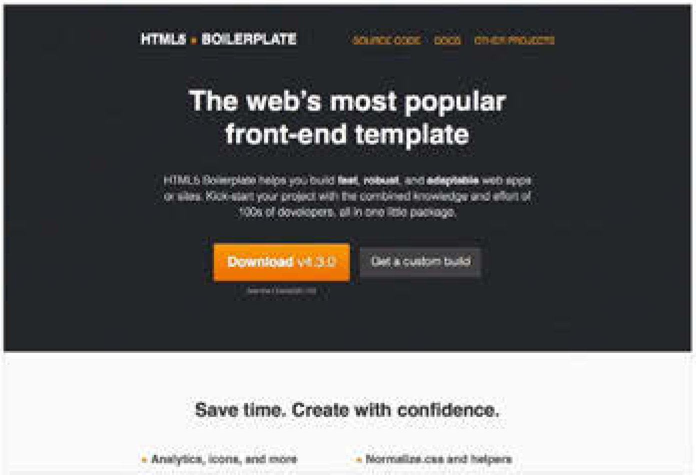
```
<input min='1' type="number" id="loanlength" value="30">
```

`type="number"` is now very well recognised. Most web browsers will now add up or down arrows and honour `"min"` and `"max"` (and even `"step"`) values when they see that input type. Using this technique is safe for old browsers too. Any input type that is not recognised becomes a normal text input. New and noteworthy input types to consider using are `type="color"`, `type="date"` and `type="email"`.

Though many browsers will highlight a selected text box, we can still help out the ones that don't:

```
 $( "input" ).focus(function(){
          $( this ).addClass("ui-state-highlight");
 }).blur(function(){
          $( this ).removeClass("ui-state-highlight");
 });
 $("#originalloanamount").focus(); $("button").button();
```

Any input field that gets focus will be highlighted using `ui-state-highlight`, a helper class defined by the jQuery UI CSS libraries. The other two commands

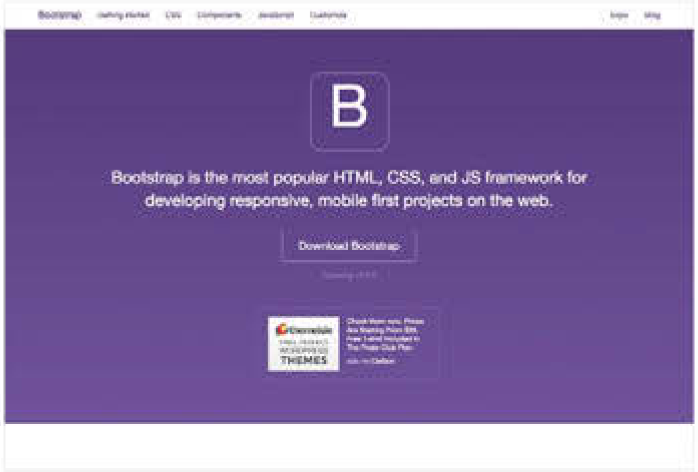


The web's most popular front-end template

HTML5 Boilerplate helps you build **fast**, **robust**, and **adaptable** web apps or sites. Kick-start your project with the combined knowledge and effort of 100s of developers, all in one little package.

Save time. Create with confidence.

- Analytics, icons, and more
- Normalize.css and helpers



Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.

Left Popular frontend template HTML5 Boilerplate provides the basis for our UI

Right Bootstrap is an alternative framework option that we could have chosen instead of jQuery UI

give the loan input field focus when the page loads (saving the user a click) and the other tells jQuery UI to style our buttons.

ADDING THE APPLICATION LOGIC

All this UI, and our application still does not calculate a mortgage. This is easily fixed. Bind to the Calculate button:

```
$( "#CalculateButton" ).click(function(){
   mtgCalc($('#originalloanamount').asNumber(),
     $('#interestrate').asNumber(), $('#loanlength').val());
```

The `asNumber()` is from our jQuery currency library loaded earlier. It parses the string, removes non-standard characters, and returns a JavaScript numeric double, suitable for our mortgage function. This finishes the calculation.

The next step is to display the result. We will create a template with Mustache.js. I prefer to keep all my templates in a separate file called `templates.js` and store them in a global variable called `Templates`. This is the code for our results templates:

```
Templates['results'] =  'Monthly Payment:
<b>{{monthly}}</b>';
```

Mustache.js takes a JavaScript object and expands its references inside a given template. There are far more powerful templating systems, but Mustache.js offers good functionality for its size, and is infinitely preferable to taking the long journey back to the server for such a simple calculation and result.

The code you need to use is as follows:

```
var result = { monthly: toMoney(mtgObj['monthly']) };
var resultshtml = Mustache.render(Templates['results'],
result);
$("#results").html(resultshtml);
```

The main function in your Mustache.js arsenal is `Mustache.render()`, which executes the template and gives back the result. Another useful feature is using an ampersand (`&`) inside your template to suppress escaping the output. Our application is now finished.

SHIP IT

Last of all, we need to minify all our code into one package for the browser. This saves the browser additional connections to fetch our application, and protects our code by making it more difficult to reverse-engineer. The tool I recommend for this task is UglifyJS. Here is a quick command to minify all the `.js` files in a directory into `main-ugly.js`:

```
rm main-ugly.js ; cat *.js | uglifyjs -nc  > main-ugly.js
```

Run a similar command on each directory you wish to minify. As a long-term strategy for code minification, I recommend taking a look at Grunt. UglifyJS also supports recursive minification, but for now, the directory-level approach is sufficient.

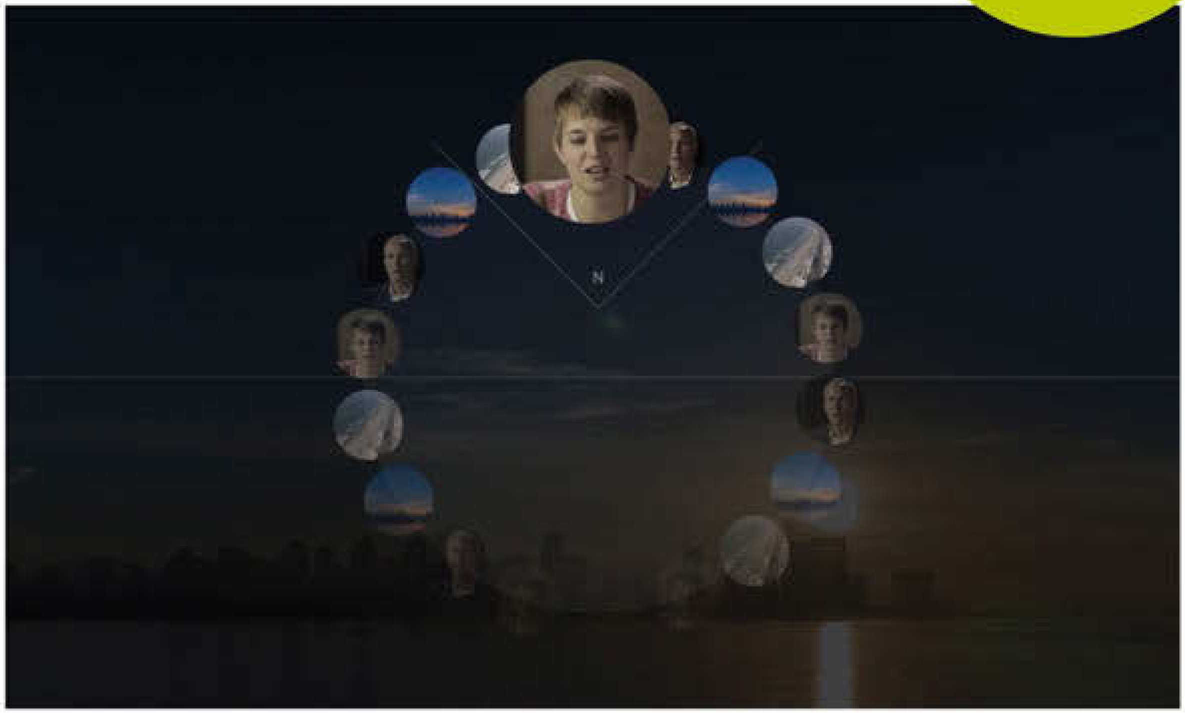
Now we need to tell the browser to load the minified files instead of the development files. We can use `filemtime()` again and let the PHP switch between normal "main-ugly.js" and our normal files depending on which is most recent. However you decide to manage it, only upload the minified version to the production server.

EXPLORE FURTHER

By leveraging HTML5 Boilerplate to get started, jQuery UI for better widgets, Mustache.js for lightweight templates, and UglifyJS to speed up the production server, we've laid the groundwork for a modern web UI. A respectable website can be built and maintained from these tools, but don't stop here. More sophisticated tools exist, and I encourage you to explore them as you optimise your workflow. ■

Download the files here!
All the files you need for this tutorial can be found at *netm.ag/radialGit-262*

ABOUT THE AUTHOR
MATTHEW KEAS
w: *mkeas.org*
t: @matthiasak
areas of expertise: CSS, HTML5, animation, JavaScript frameworks, interactive media, performance and security analysis
q: how do you deal with stressful situations?
a: I'm a fan of philosophy, and especially stoicism. It's grounding, and helps me focus on the next step – in every part of my life



USER INTERFACES

CREATE A CUSTOM USER INTERFACE

Matt Keas looks at how to develop custom, cross-platform UI controls to create unique, interactive experiences on the web

Custom UI controls are notoriously difficult to implement and can behave inconsistently on different devices. Executing custom designs requires pretty intimate knowledge of layout and positioning with CSS, browser support and a strong grasp of JavaScript. Wrapping all this up in an edgy user interface can be tiresome, tedious and taxing. In this article, I will demonstrate a UI approach for an interactive web documentary, establish a 'DIY' code ethic for this UI, and hopefully demonstrate how JS, CSS and HTML can be combined to create an original interface.

As my example, I will use an original interface I developed for a web documentary called *Facing North* (*areyoufacingnorth.com*). The main interaction of *Facing North* involves a rotating menu, which fades in when the screen is touched or the mouse is moved during video playback, and fades out again after a few seconds. *Facing North* consists of 18 videos, which can be viewed by dragging the compass in either direction via mouse or touch. The compass will track with the cursor or finger, enabling the user to select the video they want to play.

VIDEOS

Matt Keas has given a number of conference talks on his experience of building *Facing North*. You can find them all listed at: *netm.ag/talks-262*

THE APPROACH: HTML AND CSS

Creating custom UI controls requires intimate knowledge of CSS layouts and positioning. Focusing on layout, animations and pseudo-classes produces effective, responsive UIs. Here is the HTML for the *Facing North* compass menu:

```
<div class="container">
        <div class="compass">
                <!--
```

18 of these will be created by JavaScript:

```
      <img class="orbiter" src="...">
      -->
      <div class="north-indicator"></div>
    </div>
  </div>
```

The .container element adds the transparent black overlay and can be sized using vh and vw . Safari on iOS currently has some support issues, so it's best to recalculate the CSS with JavaScript onresize or use % widths. The design includes a horizontal 'range' across the middle screen, which can be accomplished with a CSS pseudo-element.

The .compass element is highlighted by a white, circular border. The .compass must be small enough to fit on the screen, it must sit absolutely centred on the .container , and it must be perfectly circular. It's possible to maintain a square box model by using an ::after pseudo-element that has padding-top:100% .

The .orbiter elements are simply <img> tags that 'orbit' around the compass. These will be created in our JavaScript, and using some maths we can ensure any number of orbiters will distribute evenly. The .north-indicator element is positioned at the top of the compass, denoting north. The primary problem in the UI is the transforms applied to each .orbiter element. This is handled in the JavaScript.

Layout, animations and pseudo-classes help produce effective, responsive UIs

THE GLUE: JAVASCRIPT

There are a few major components of the JavaScript for *Facing North*'s compass UI:

- A prototype (JavaScript 'class') to handle logic for the entire compass UI
- A prototype (JavaScript 'class') to handle logic for each orbiter
- Polyfills (to help support older browsers)

The JavaScript initialises onload with:

```
window.onload = app;
function app() {
  rAFPolyfill();
  window.vendorPrefix = getVendorPrefix();
```

▶

RESOURCES

EXPLORE FURTHER

Interactive web documentaries are intended to be viewed and experienced in a browser environment, typically involving some mix of interaction and audio-visual feedback. There is no de-facto standard, book or blog on how to code or implement interactive media, which means these types of projects present something of a blank canvas. Are you ready to start painting?

For more info on what JavaScript framework to use, check out:

- DailyJS' series on DIY jQuery (*dailyjs.com/framework.html*)
- Addy Osmani's book, *Learning JavaScript Design Patterns* (*netm.ag/osmani-262*)
- The jQuery source code (*netm.ag/sourcecode-262*)
- The Backbone source code (*backbonejs.org/docs/backbone.html*)

It takes a lot of extra reading and work to fully explore all these items, but a few hours of dedicated practice, a notepad and some great notes can you really boost your skills.

Another great angle is to look at more examples of interactive web documentaries. Here are some of my latest favourites:

- *Clouds Over Cuba* (*cloudsovercuba.com*) – an interactive and historic documentary on the Cuban missile crisis
- Pharrell's *24 Hours of Happy* (*24hoursofhappy.com*) – an interactive music video with a circular UI
- *Hollow* (*hollowdocumentary.com*) – an interactive web documentary and immersive multimedia collage centred around a McDowell County in West Virginia
- *This Place* (*thisplacejournal.com*) - a web documentary with accompanying images and posts about the Oregon coastline

Finally, to look deeper into *Facing North* (*areyoufacingnorth.com*), you can check out my talk about building it (*netm.ag/keas-262*) and a further demo on its layout (*netm.ag/layering-262*).

```
        var compass = new Compass(18, document.
querySelector('.compass'));
    }
```

In app(), there is a polyfill added for requestAnimationFrame(), we create a new instance of the Compass 'class', and getVendorPrefix() is called to retrieve CSS prefixes of the browser – for example -webkit-, -ms or -moz-. The rest of the JavaScript is run through the constructor new Compass(), which takes num orbiters to create and distribute around the compass UI, and a reference to the compass UI DOM element.

The compass constructor is quite simple:

```
function Compass(num, compassElement) {
  this.rotation = 0;
  this.orbiters = this.addOrbiters(num, compassElement);
  this.compassElement = compassElement;
  this.rotate();
  this.handleInteractionEvents();
}
```

Most of the animation and interaction is handled by handleInteractionEvents(). addOrbiters() just creates num Orbiter() objects and stores them in an array:

```
Compass.prototype.addOrbiters = function(num, compass)
{
  var i = num,
    orbiters = [];

  while (typeof i === "number" && i--) {
    orbiters.push(new Orbiter(compass, i));
  }

  return orbiters;
}
```

The Orbiter() constructor in the live version pulls JSON data from a server; the following is just a contrived example created for this article, which creates <img> tags with JavaScript:

```
function Orbiter(compass, index) {
  var img = document.createElement('img');
  img.className = "orbiter";
  img.src = "./images/screen" + (index % 4 + 1) + ".png";
  this.img = img;
  compass.appendChild(img);
}
```

Compass.rotate() is the function used to animate the position of each Orbiter() during interaction:

It doesn't take much extra code to handle two event types – mouse and touch

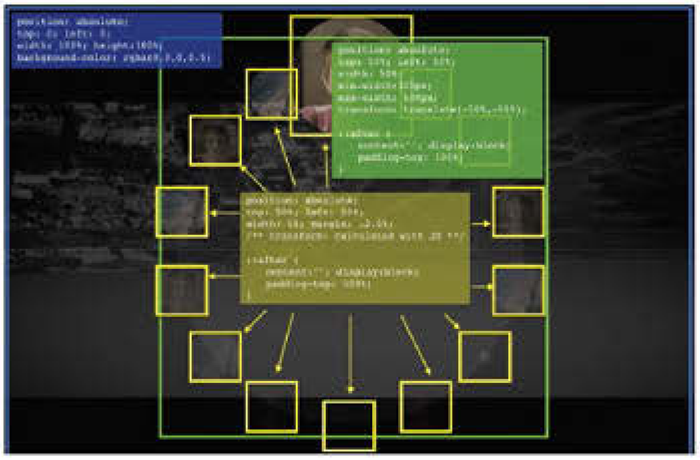
```
Compass.prototype.rotate = function(delta) {
  var self = this;
  for (var i = 0, len = self.orbiters.length; i < len; i++) {
     self.orbiters[i].setTransform(i, self.compassElement.
offsetWidth, delta || 0, ~~(360 / self.orbiters.length));
  }
}
```

Orbiter.rotate() has a few things worth noting here:

1 i: the spot in place (an integer between 0 to 17 if there are 18 Orbiter() objects)
2 self.compassElement.offsetWidth: the diameter of the compass, in pixels
3 delta || 0: the current rotation (in degrees, between 0 and 359) to set each Orbiter(). Orbiter()s with an i greater than 0 will be rotated around further to ensure even distribution

Below left This image denotes the different elements on the screen

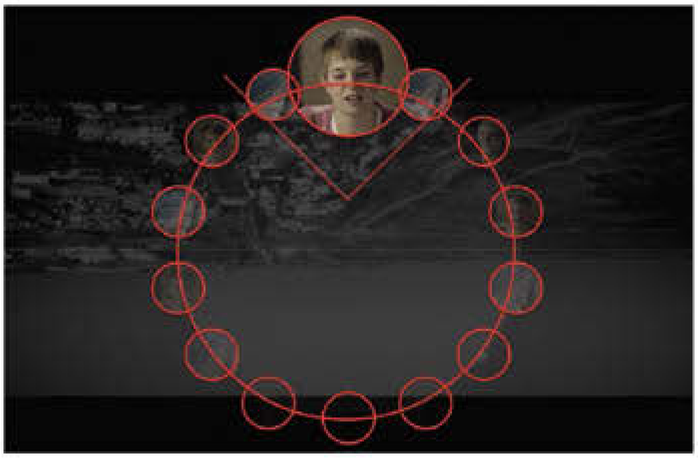
Below right This demonstrates the positioning of the .orbiters and the .compass elements



```
position: absolute;
top: 0; left: 0;
width: 100%; height:100%;
background-color: rgba(0,0,0,0.5);
```

```
position: absolute;
top: 50%; left: 50%;
width: 50%;
min-width: 320px;
max-width: 480px;
transform: translate(-50%,-50%);

:after {
  content:''; display:block;
  padding-top: 100%;
}
```

```
position: absolute;
top: 50%; left: 50%;
width: 1%; margin: -2.5%;
/** transform: calculated with JS **/

:after {
  content:''; display:block;
  padding-top: 100%;
}
```



4 `~~(360 / self.orbiters.length)` : the degrees by which to spread out each `Orbiter()` . For example, for 18 orbiters, this will be 360/18, or 20°. `~~` rounds a decimal number to the nearest integer

The `Orbiter.setTransform()` code can be found online at the GitHub repo (*netm.ag/radialGit-262*).

The rest of the `Compass()` constructor involves initialising touch and mouse event handlers:

```
Compass.prototype.handleInteractionEvents = function() {
  var self = this;

  function onTouchAndMouseStart(e) { ... }
  function onTouchAndMouseEnd(e) { ... }
  function handleDrag(e) { ... }

  $.on('mousedown touchstart', onTouchAndMouseStart,
this.compassElement);
  $.on('mouseup touchend', onTouchAndMouseEnd,
window);
}
```

Notice the three functions for handling touch and mouse events:

1 `onTouchAndMouseStart()` : triggered on `mousedown` or `touchstart` when the interaction with the compass UI starts
2 `onTouchAndMouseEnd()` : triggered on `mouseup` or `touchend` when interaction has stopped and stops animating the compass UI
3 `handleDrag()` : triggered on `mousemove` or `touchmove` when the compass UI is 'dragged' and animates the compass UI to 'track' with the user's mouse or finger

The code for these handlers, `onTouchAndMouseStart()` , `handleDrag()` and `onTouchAndMouseEnd()` can be found online at the Github repo.

FINAL THOUGHTS

Through a strategic approach, we have created a reusable interface pattern. It didn't take much extra code to handle two separate event types (mouse and touch). Doing as much animation as possible via CSS helped optimise rendering, and the 'drag' events were handled only when the 'start' events took place, to optimise CPU resources. For an in-depth look, visit the GitHub repo (*netm.ag/radialGit-262*).

The final product benefits from a DIY code ethic. Without researching, exploring and prototyping the 'hard parts' (read: the maths and UI approaches), I would not have been able to make this work as smoothly as I did. The maths was both the most difficult and most rewarding part of the UI.

IN-DEPTH

ROTATING ELEMENTS

CSS Transforms can facilitate some really edgy designs. *Facing North*'s compass UI, for instance, involves orbiting elements that animate and revolve around the centre of a circle. Let's assume our container element (with class `.compass`) has `border-radius: 50%` and is horizontally and vertically centred on the screen. Any `.orbiter` element on the interface can rotate around the edge of the circle.

This is easily done with CSS transforms, as well. Consider the following HTML:

```
<div class="compass">
  <img class="orbiter" src="./images/screen1.png">
</div>
```

... and CSS:

```
.compass {
  width: 500px; height: 500px;
  border-radius: 50%;
  position: absolute;
  top: 50%; left: 50%;
  transform: translate(-50%, -50%) translateZ(0);
}

.orbiter {
  position: absolute;
  width: 15%; margin: -7.5%;
  top: 50%; left: 50%;
  border-radius: 50%;
}
```

This puts the `.orbiter` smack dab in the centre of the `.compass`. We can easily rotate the `.orbiter` and then move it 250px (the radius of the `.compass`) with this extra line:

```
transform: rotate(25deg) translate(-250px);
```

Unfortunately, you'll notice that the `.orbiter` doesn't keep its vertical orientation. The more we rotate it round, the more it will flip upside-down. To fix this, we can add an extra and equal rotation to the `.orbiter`:

```
transform: rotate(25deg) translate(-250px) rotate(-25deg);
```

Rotating 25 degrees, translating the `.orbiter`, and the rotating back keeps the `.orbiter` in the same spot, but reorients the `.orbiter` element so it's be vertically aligned again. This is one nifty trick we can do with transforms.

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/touch-248*

ABOUT THE AUTHOR

PATRICK H LAUKE

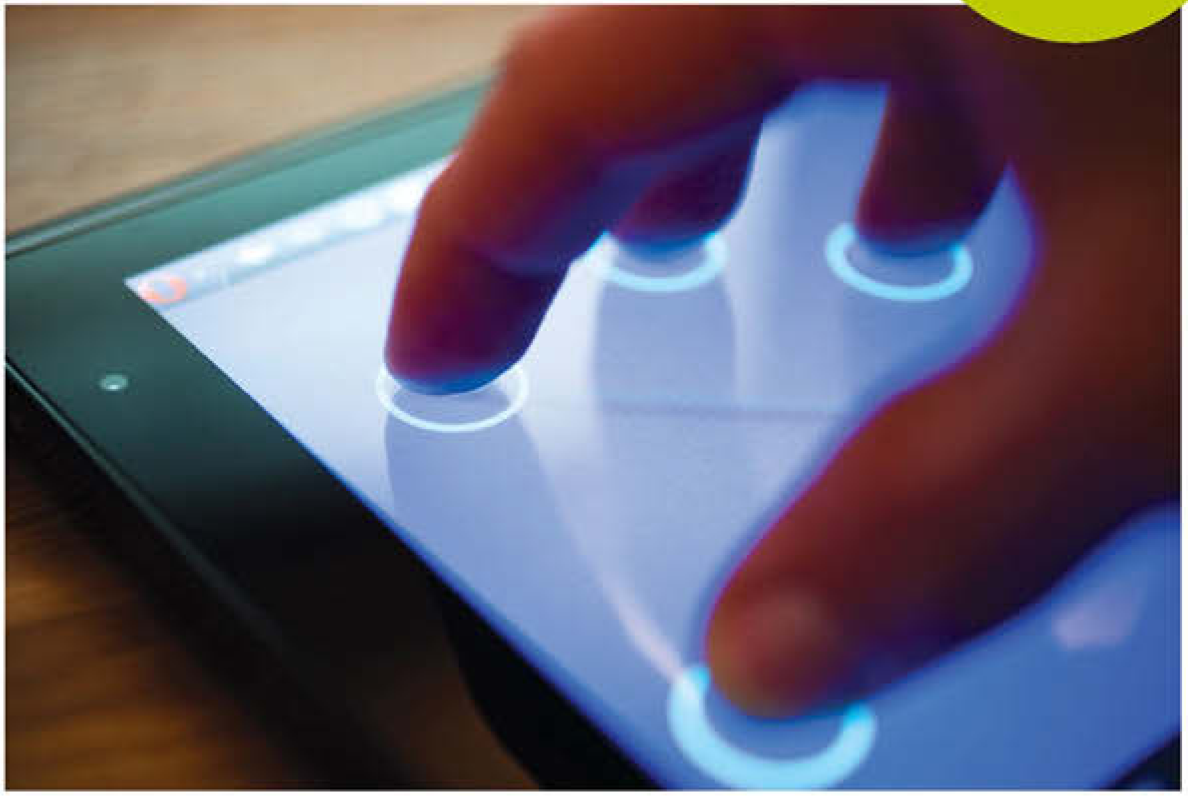
w: *splintered.co.uk*

t: @patrick_h_lauke

areas of expertise: HTML5, JavaScript, accessibility

q: what's the oddest thing you've ever seen on a designer's desk?

a: A four-piece matryoshaka doll with all the band members from U2



TOUCH

MAKE YOUR SITE WORK ON TOUCH DEVICES

Patrick H Lauke demonstrates how to make your site work on touch devices, in this introduction to handling touch events in JavaScript

BROWSER SUPPORT
TOUCH EVENTS

Desktop	Mobile/tablet
22	11
6	2.1
No	11
15	29
No	24

Touchscreens on mobile phones, tablets and touch-enabled laptops and desktops open a whole new range of interactions for web developers. In this article, we'll look at the basics of how to handle touch events in JavaScript. In the supporting files (*netm.ag/touch-248*) you'll find the example demos that I refer to throughout the tutorial.

With the rise of touchscreens, one of the fundamental questions from developers has been: what do I need to do to make sure my website or app works on touch devices? Surprisingly, in most cases, the answer is: nothing at all. By default, mobile browsers are designed to cope with the large amount of existing websites that weren't developed specifically for touch. Not only do these browsers work well with static pages, they also handle sites that provide dynamic interactivity through mouse-specific JavaScript, where scripts have been hooked into events like `mouseover`. To this end, browsers on touch-enabled devices trigger simulated, or synthetic, mouse events. A simple test page (see `example1.html` in the tutorial files) shows that, even on a touch device, tapping a button fires the following sequence of events: `mouseover > (a single) mousemove > mousedown > mouseup > click`.

These events are triggered in rapid succession, with almost no delay between them. Also note the single 'sacrificial' `mousemove` event, which is included to ensure any scripts that may be listening to mouse movement are also being executed at least once. If your website is set to react to mouse events, its functionality will (in most cases) still work without requiring modifications on touch devices.

As good as the fallback to simulated mouse events is, there are, however, still situations where purely relying on mouse-specific scripts may result in a suboptimal experience.

DELAYED CLICK EVENTS

When using a touchscreen, browsers introduce an artificial delay (in the range of about 300ms) between a touch action and the time the actual click event is fired. This delay allows users to double-tap (for instance, to zoom into a page) without accidentally activating any page elements (see example2.html). This delay can be a problem if you want to create a web app that feels snappy. For regular web pages, this is unlikely to be an issue as this is the default behaviour users understand from most sites.

TRACKING FINGER MOVEMENTS

As we already saw, the synthetic mouse events dispatched by the browser also include a mousemove event. This will always be just a single mousemove. In fact, if users move their finger over the screen too much, synthetic events will not be fired at all, as the browser interprets the movement as a gesture, such as scrolling. This is a problem if your site relies on interactions involving mouse movements (such as drawing applications or HTML-based games), as simply listening to mousemove won't work on touch devices. To illustrate this, let's create a simple canvas -based application (see example3.html). Rather than the specific implementation, we're interested in how the script is set to react to mousemove :

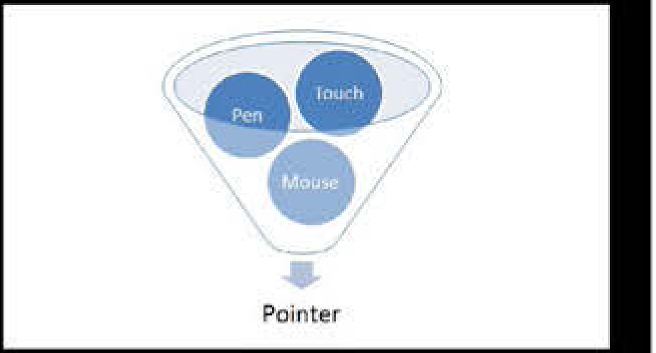
```
var posX, posY;
...
function positionHandler(e) {
  posX = e.clientX;
  posY = e.clientY;
}
...
canvas.addEventListener('mousemove', positionHandler,
false );
```

If you try out example three (in the tutorial files) with a mouse, you'll see that the position of the pointer is continuously tracked as you move over the canvas . On a touch device, you'll notice it won't react to finger movements; it only registers when a user taps on the screen, which will fire that single synthetic mousemove event.

▶

FOCUS ON

POINTER EVENTS



While almost all browsers support the touch event model, for the longest time the exception has been Internet Explorer. Versions up to IE9 lack support for any low-level touch interaction. However, IE10 introduced Microsoft's Pointer Events, its own event model that unifies 'pointer' devices (such as a mouse, stylus or touch) under a single new class of events, making it possible to write input-agnostic code. Browsers that support pointer events still generate synthetic mouse events for legacy sites, which are dispatched 'inline' (unlike touch events, where fallbacks are fired afterwards). The delay happens after the last mouse event and just before the final click : pointerover > mouseover > pointerdown > mousedown > pointermove > mousemove > pointerup > mouseup > pointerout > mouseout > [300ms delay] > click .

The simulated mouse events also can't be suppressed with preventDefault() . Instead, pointer events have a new CSS property, touch-action , which defines the default behaviour for the browser when handling touch interactions.

```
#foo { touch-action: none; } /* suppress default behaviour */
```

Pointer Events recently became a W3C Proposed Recommendation, and Firefox is working on its own implementation. Unfortunately, despite being directly involved in the refinement of the specification at the W3C, Google decided it would not roll out Pointer Events in Chrome, and all signs suggest Apple has no intention of including this new event model in Safari. At the same time, IE 11 in Windows Phone 8.1 has – for compatibility reasons – recently included support for actual touch events, in parallel to Pointer Events.

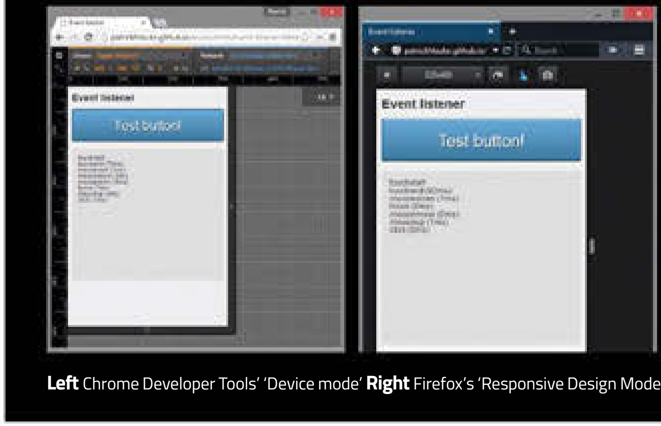
Pointer Events do provide a more powerful model to create 'input-agnostic' code (while also allowing for code that can easily distinguish, and react differently to, different input modalities). If you do want to take advantage of Pointer Events already, without sacrificing cross-browser compatibility, there are polyfill libraries such as Microsoft's HandJS (*handjs.codeplex.com*). *[For more on pointer events, flip to page 181].*

TEST TIPS

TESTING AND DEBUGGING TOUCH EVENTS

The best way to test and debug your touch-specific code is, obviously, on a touch-capable device. Depending on your development workflow, you may want to debug your code directly on your non-touch desktop or laptop. Chrome and Firefox offer device emulation functionality, which includes the ability to simulate touch events.

In the early examples of this tutorial, we used a custom test page to check the order in which events are fired. A useful utility function in the Chrome DevTools console that achieves the same effect by listing out any events that are fired on a particular element is `monitorEvents()` (*netm.ag/events-248*). With this function, it's possible to pinpoint event dispatch issues without having to modify your existing code or throw `console.log()` all over the place. As this function is Chrome-specific, it's a good idea to test in other browsers as well.



Left Chrome Developer Tools' 'Device mode' **Right** Firefox's 'Responsive Design Mode'

GOING DEEPER

To work around these issues, we need to go to a lower abstraction level. Touch events were first introduced in Safari for iOS 1.0, and, following widespread adoption in (almost) all other browsers, were retrospectively standardised in the W3C Touch Events specification. The new events provided by the touch events model are: `touchstart`, `touchmove`, `touchend` and `touchcancel`. The first three are the touch-specific equivalent to the traditional `mousedown`, `mousemove` and `mouseup` events. On the other hand, `touchcancel` is fired when a touch interaction is interrupted or aborted. For example, when a user moves their finger outside of the current document and into the actual browser interface.

Looking at the order in which both touch and synthetic mouse events are dispatched for a tap (see `example4.html` in the tutorial files), we get the following sequence: `touchstart > [ touchmove ]+ > touchend > mouseover > (a single) mousemove > mousedown > mouseup > click`.

First, we get all the touch-specific events: `touchstart`, zero, or more `touchmove` (depending on how cleanly the user manages to tap without moving the finger during contact with the screen) and `touchend`. After that, the browser will fire the synthetic mouse events and the final `click`.

To determine if a browser supports touch events, we can use JS feature detection

FEATURE DETECTION

To determine if a particular browser supports touch events, we're able to use a simple bit of JavaScript feature detection:

```
if ('ontouchstart' in window) {
 /* browser with Touch Events support */
}
```

This snippet works reliably in modern browsers. Older versions have a few quirks and inconsistencies that require you to jump through various different detection strategy hoops. If your application is targeting older browsers, try Modernizr (*modernizr.com*) and its various touch test approaches, which smooth over most of these issues.

When conducting this sort of feature detection, we need to be clear what we're testing. The prior snippet only checks for the browser's capability to understand touch events and shouldn't be used as a way of checking if the current page is being viewed on a touchscreen-only device. There is a new class of hybrid devices, which feature both a traditional laptop or desktop form factor (mouse, trackpad, keyboard) and touchscreen, meaning it's no longer an either-or proposition as to whether the user will interact with our site via a touchscreen or a mouse.

WORKING AROUND THE CLICK DELAY

If we test the sequence of events dispatched by the browser on a touch device and include some timing information (see `example5.html` in the downloadable tutorial files), the 300ms delay is introduced after the `touchend` event: `touchstart > [ touchmove ]+ > touchend > [300ms delay] > mouseover > (a single) mousemove > mousedown > mouseup > click`.

So, if our scripts are currently set to react to `click` events, we can remove the sluggish browser behaviour and prevent the default delay. We do this by reacting to either `touchend` or `touchstart` – the latter for interface elements that need to fire immediately when a user touches the screen.

Once again, we must be careful not to make false assumptions about touch event support and actual touchscreen use. Here's one of the common performance tricks that's quite popular and often mentioned in mobile optimisation articles.

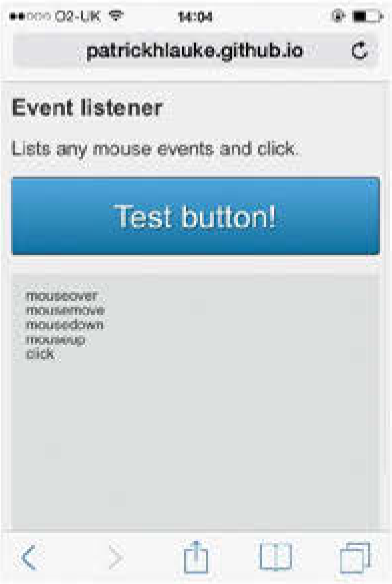
```
/* if touch supported, listen to 'touchend', otherwise 'click' */
var clickEvent = ('ontouchstart' in window ? 'touchend' :
'click');
blah.addEventListener(clickEvent, function() { ... });
```

Although this script is well-intentioned, the mutually-exclusive approach of listening to either `click` or `touchend` depending on browser support for touch events will cause problems on hybrid devices as it will immediately shut out any interaction via mouse, trackpad or keyboard.

For this reason, a more robust approach would be to listen to both types of events:

```
blah.addEventListener('click', someFunction, false);
blah.addEventListener('touchend', someFunction, false);
```

The problem with this approach is that our function will be executed twice: once as a result of `touchend`, and a second time when the synthetic mouse events and `click` are being fired. One way to work around this is to suppress the fallback mouse events entirely by using `preventDefault()`. We can also prevent code repetition by simply making the `touchend` event trigger the actual `click` event.

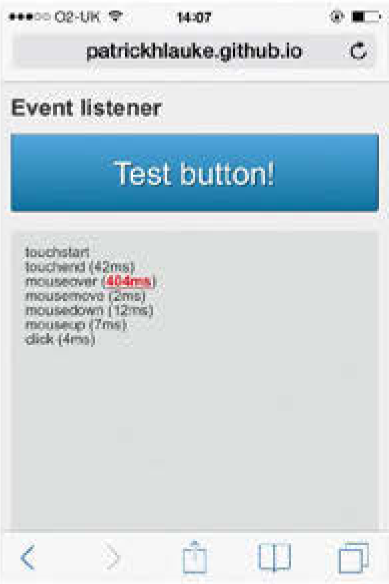


Event listener

Lists any mouse events and click.

```
mouseover
mousemove
mousedown
mouseup
click
```

On fire (above left) Mouse events and `click` fire even for a touchscreen tap



Event listener

```
touchstart
touchend (42ms)
mouseover (404ms)
mousemove (2ms)
mousedown (12ms)
mouseup (7ms)
click (4ms)
```

On Safari (above right) Events fired in iOS Safari, and showing the delay after `touchend`

```
blah.addEventListener('touchend', function(e) {
  e.preventDefault();
  e.target.click();
}, false);
blah.addEventListener('click', someFunction, false);
```

There's a catch. When using `preventDefault()`, we also suppress any other default behaviour of the browser. If we apply it directly to `touchstart` events, any other functionality like scrolling, long click or zooming will be suppressed as well. Sometimes, this may be desirable, but generally this method should be used with care. Also note that the above example code hasn't been fully optimised. For a robust implementation, check out FTLabs' FastClick by visiting: *github.com/ftlabs/fastclick*.

Browsers have also started to implement specific tricks to attempt to 'optimise away' the click delay in specific situations – for instance, on sites that define specific viewport characteristics such as:

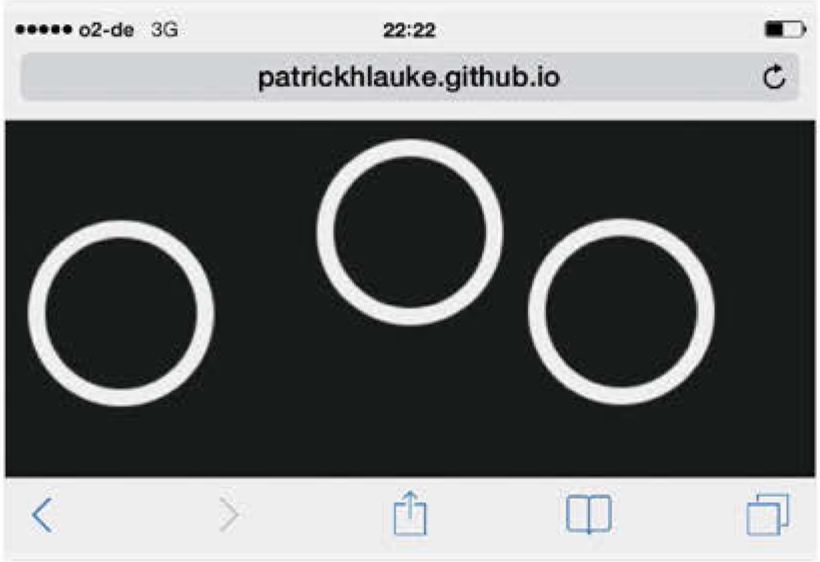
```
<meta name="viewport" content="user-scalable=no">
```

For a basic breakdown, see *netm.ag/supressing-265*. However, none of these more recent tricks currently work in Safari/WebView on iOS – so the use of `preventDefault()` in the previous code (or the use of helper libraries like FastClick) is still the most reliable cross-browser option.

TRACKING MOVEMENT

Armed with our touch events knowledge, let's go back to the tracking example (`example3.html`) and see how we can modify it to also track finger movements on a touchscreen. Before looking at the specific

▶



Multiple touchpoints The final example in the tutorial files shows tracking the of multi-touch interactions

▶ changes needed in our script, we need to understand how touch events differ from mouse events.

ANATOMY OF A TOUCH EVENT

In accordance with the DOM Level 2 Events Specification (*netm.ag/dom-248*), functions that listen to mouse events receive a `MouseEvent` object as parameter. This object includes coordinate properties such as `clientX` and `clientY`, which our script (`example3.html` in the tutorial files) uses to determine the current mouse position. For example:

```
interface MouseEvent : UIEvent {
  readonly attribute long             screenX;
  readonly attribute long             screenY;
  readonly attribute long             clientX;
  readonly attribute long             clientY;
  readonly attribute boolean          ctrlKey;
  readonly attribute boolean          shiftKey;
  readonly attribute boolean          altKey;
  readonly attribute boolean          metaKey;
  readonly attribute unsigned short   button;
  readonly attribute EventTarget      relatedTarget;
  void         initMouseEvent(...);
};
```

Touch events extend the approach taken by mouse events. As such, they pass on a `TouchEvent` object that's very similar to a `MouseEvent`, but with one crucial difference: as modern touchscreens generally support multi-touch interactions, `TouchEvent` objects don't contain individual coordinate properties. Instead, the coordinates are contained in separate `TouchList` objects:

```
interface TouchEvent : UIEvent {
  readonly attribute TouchList touches;
  readonly attribute TouchList targetTouches;
  readonly attribute TouchList changedTouches;
  readonly attribute boolean   altKey;
  readonly attribute boolean   metaKey;
  readonly attribute boolean   ctrlKey;
  readonly attribute boolean   shiftKey;
};
```

As we can see, a `TouchEvent` contains three different `TouchList` objects:

- `touches`: includes all touch points that are currently active on the screen, regardless of whether or not it's directly related to the element we registered the listener function for
- `targetTouches`: only contains touch points that started over our element – even if the user moved their finger outside of the element itself
- `changedTouches`: includes any touchpoints that changed since the last touch event – in effect, which change (number of active touchpoints, or position of those points) caused the touch event itself to fire

We need to modify our listener function so that it reacts both to mouse and touch events

Each of these represents an array of individual `Touch` objects. Here we find the coordinate pairs like `clientX` and `clientY` that we're after:

```
interface Touch {
  readonly attribute long        identifier;
  readonly attribute EventTarget target;
  readonly attribute long        screenX;
  readonly attribute long        screenY;
  readonly attribute long        clientX;
  readonly attribute long        clientY;
  readonly attribute long        pageX;
  readonly attribute long        pageY;
};
```

FINGER TRACKING

Let's return to our `canvas`-based example. First, we need to modify our listener function so it reacts both to mouse and touch events. In the first instance, we're only interested in tracking the movement of a single touch point that originated on our canvas. So, we'll just grab the `clientX` and `clientY` coordinates from the first object in the `targetTouches` array:

```
var posX, posY;
function positionHandler(e) {
 if ((e.clientX)&&(e.clientY)) {
  posX = e.clientX;
  posY = e.clientY;
 } else if (e.targetTouches) {
  posX = e.targetTouches[0].clientX;
  posY = e.targetTouches[0].clientY;
  e.preventDefault();
 }
}
canvas.addEventListener('mousemove',  positionHandler,
false );
canvas.addEventListener('touchstart', positionHandler, false
);
canvas.addEventListener('touchmove',  positionHandler, false
);
```

Testing the modified script (see `example6.html` in the tutorial files) on a touchscreen device, you'll see tracking a single finger movement now works reliably. If we want to expand our example to also work for multi-touch, we'll need to modify our original approach.

Instead of a single coordinate pair, we'll consider a whole array of coordinates, which we'll process in a loop. This will allow us to track single mouse pointers as well as any multi-touch finger movements a user makes (see `example7.html` in the supporting files):

```
var points = [];
function positionHandler(e) {
 if ((e.clientX)&&(e.clientY)) {
  points[0] = e;
 } else if (e.targetTouches) {
  points = e.targetTouches;
  e.preventDefault();
 }
}
function loop() {
 ...
 for (var i = 0; i<points.length; i++) {
  /* Draw circle on points[0].clientX / points[0].clientY */
  ...
 }
}
```

PERFORMANCE CONSIDERATIONS

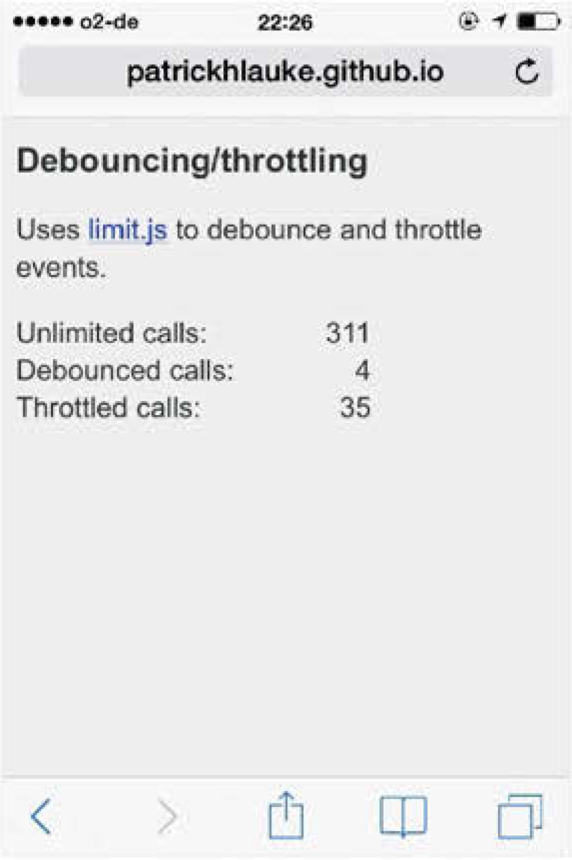
As with `mousemove` events, `touchmove` can fire at a high rate during any finger movements. It's advisable to avoid executing intensive code, such as complex calculations, or even entire drawing operations, for each move event. This is important for older, less performant touch devices. In our example, we do the absolute minimum by simply storing the latest array of mouse or touch point coordinates. The code to actually redraw our canvas is executed independently in a separate loop called via `setInterval`.

If the number of events that your script needs to process is still too high, it may be worth debouncing or throttling these events further with solutions like limit.js (*netm.ag/limit-248*).

CONCLUSION

Although, by default, browsers on touch-capable devices do a reasonable job of handling mouse-specific scripts, there are still situations where it may be necessary to further tweak our code specifically for touch interactions.

Throughout this project tutorial, we've looked at the basics of how to handle touch events in JavaScript. Hopefully this tutorial has given you a solid introduction into why touch events are necessary, as well as a foundation to build on for how they can be used to make your websites and applications work well on touch devices.



Debouncing/throttling

Uses limit.js to debounce and throttle events.

Unlimited calls:	311
Debounced calls:	4
Throttled calls:	35

No limits Debouncing and throttling touchmove events in action with limit.js

SLIDEDECK

GETTING TOUCHY

See Patrick H Lauke's slides from his workshop, 'Getting Touchy: An Introduction to Touch and Pointer Events' from JavaScript Days in Berlin: *netm.ag/touchy-265*

ABOUT THE AUTHOR

JONATHAN FIELDING

w: *jonathanfielding.com*

t: @jonthanfielding

areas of expertise: jQuery, JavaScript, open source

q: what's the weirdest present you've ever been given?

a: an AirZooka, which is like a cannon that fires a burst of air. My kids loved it though

BROWSER SUPPORT MATCHMEDIA API

Desktop	Mobile/tablet
9	10
6	3
10	12.1
12.1	30
5.1	25

RWD

LEARN TO MASTER RESPONSIVE JAVASCRIPT

Jonathan Fielding shows that handling JavaScript across responsive states doesn't need to be a minefield

When we think of responsive development, the first thing that comes to mind is how we can use media queries to change the look and feel of our site depending on rules defined by the query. Media queries are conditional, with CSS being applied if the browser is meeting the condition. A visual change to a site may also require a change to how the site functions. This is where we start using JavaScript.

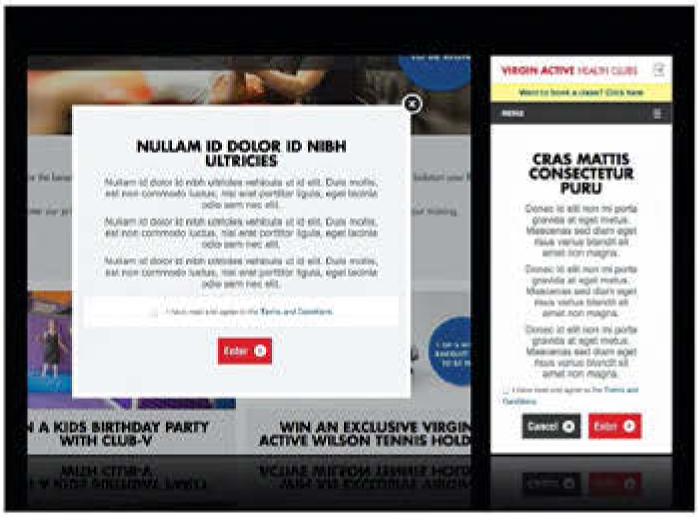
There are many examples of where we may want to change functionality; for example, where on desktop we want to show content in a modal window. This provides a good, usable experience on desktop. However, on mobile devices modal windows often look cramped and provide a poor user experience. The best option is to disable the modal functionality on mobile. It's here that CSS can fall short, and where JavaScript can be used to pick up the work.

Unfortunately, adding JavaScript that manages the change of functionality based on the responsive state of your site can be trickier than simply using a media query. This is because we need to gracefully handle the transition between states. With CSS media queries, the styles are just turned on or off, or overridden when we're in different states, but, with JavaScript, we must handle both the conditional logic and the turning on or off of functionality ourselves.

The browsers don't leave us completely unaided as newer browsers feature the matchMedia API. The matchMedia API can watch to see if a condition is met. If it is, it will fire a method. Conditions are in the form of the same media queries that we are already familiar with from CSS. To see how we can use the matchMedia API, we can take a look at the following example that logs to the browser console when we enter and leave the mobile state:

```
var mql = window.matchMedia("screen and (max-
width:768px)");
mql.addListener(function(e){
    if(e.matches){
        console.log('enter mobile');
    }
    else{
        console.log('leave mobile');
    }
});
```

We set up a query list using `window. matchMedia` and set up query notifications by calling the `addListener()` method passing our callback. When the media query is matched or unmatched, our listener method is called and we can handle the change of state.


NULLAM ID DOLOR ID NIBH ULTRICIES
Enter
A KIDS BIRTHDAY PARTY WITH CLUB-V
WIN AN EXCLUSIVE VIRGIN ACTIVE WILSON TENNIS HOLD
VIRGIN ACTIVE HEALTH CLUBS
CRAS MATTIS CONSECTETUR PURU
Cancel
Enter

User experience Open a new page on mobile instead of opening a lightbox

Browser support for the matchMedia API is good with the exception of Internet Explorer, as the earliest version to support it is Internet Explorer 10. If you want to use the matchMedia API, you have the option to include a polyfill for the matchMedia API, which enables support for the matchMedia API in older versions of Internet Explorer.

The main limitation of this API is that it handles the switching between states. However, it doesn't handle having a resize event for each state, which may be required for some functionality. Typically, this has led to me ignoring the matchMedia API and, instead, rolling my own solution based on using the browser resize event. This typically would look something like this:

> The main limitation of the matchMedia API is it handles the switching between states

```
var stateManager = (function () {
    var state = null;
    var resizePage = function () {
        if ($('body').width() < 768) {
            if (state !== "mobile") { displayMobile(); }
resizeMobile();
        }
        else {
            if (state !== "desktop") { displayDesktop(); }
resizeDesktop();
        }
    };
```

▶

★ FOCUS ON

MANAGING RESPONSIVE STATES WITH ENQUIRE.JS

Enquire.js (*wicky.nillia.ms/enquire.js*) describes itself as "awesome media queries in JavaScript". It is a JavaScript library, which allows you to add callbacks to the matching an unmatching of media queries. The methodology for using Enquire.js is to register a media query, along with any matching and unmatching handlers you wish to add.

When a media query is matched (or unmatched), the correct handler will fire. This means that when you enter a state you can apply any JavaScript changes you need to apply, upon leaving the state you can do your clean up.

Enquire.js allows you to use the same media queries in your JavaScript as you are using in your CSS. Not only does this mean you can use complex queries, it also allows you to easily pair up your CSS style changes with your JS functionality changes.

Enquire.js is built upon the matchMedia API we discussed in the main article. This has the benefit of the media query being parsed in the browser itself, not in the library. The disadvantage is that, being built on the matchMedia API, we will need to include a polyfill for the matchMedia API in order to support older browsers.

Enquire.js library This library makes the matchMedia API more simple to use

```
    var displayMobile = function () {
state = "mobile";
console.log("enter mobile");
    };
    var displayDesktop = function () {
state = "desktop";
console.log("enter desktop");
    };
    var resizeMobile = function () {
        console.log("resizing mobile");
    };
    var resizeDesktop = function () {
        console.log("resizing desktop");
    };
    return {
        init: function () {
            resizePage();
            $(window).on('resize', resizePage);
        }
    };
} ());
stateManager.init();
```

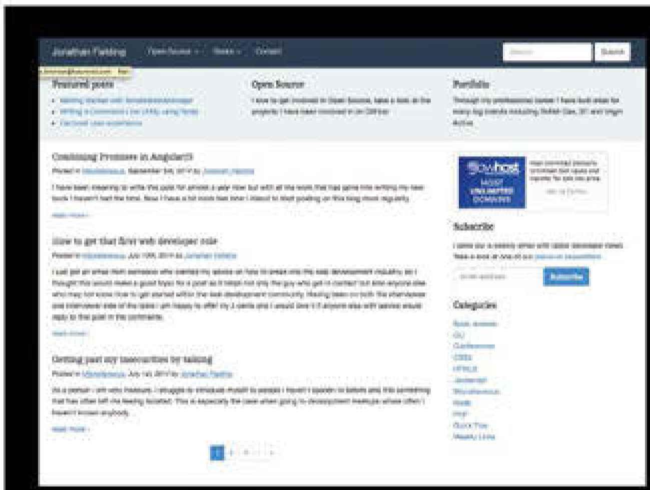
Here we added a resize event to the window that checks the current state of the browser. As the browser is resized, we check the width of the page and determine whether we're in the mobile or desktop state. Once this has been determined we check whether we're already in the state. If we are, we simply fire a state resize method. Otherwise, we fire an enter state method. Simple. However, if you have more than a couple of states, this method has the potential to become very unwieldy.

This is when I usually start looking at JavaScript libraries to see if there's something that makes things simpler.

LOOKING AT LIBRARIES

So far, we've looked at how we can simply write our own JavaScript to handle responsive states. However,

Simple states SSM is designed to target responsive states across devices



RESOURCES

FURTHER READING

There are several resources to help you learn more about writing JavaScript that responds to the state of the browser:

Try the following three articles from the Mozilla Developer Network (MDN): '**Testing Media Queries**' (*netm.ag/testing-250*), which provides an introduction on how to test media queries in JavaScript; '**MediaQueryList**' (*netm.ag/mediaquerylist-250*), which provides documentation on the MediaQueryLists API in JavaScript; and '**Window.matchMedia**' (*netm.ag/window-250*), which provides documentation on the window.matchMedia API in JavaScript.

Other useful resources include:

SimpleStateManager *simplestatemanager.com*
The site for the SimpleStateManager library includes examples on how to get started with Enquire.js along with full documentation of the API. The SimpleStateManager also includes a list of all the official plugins that are available for SimpleStateManager.

Enquire.js *wicky.nillia.ms/enquire.js*
The site for the Enquire.js library includes examples on how to get started with Enquire.js along with full documentation.

Paul Irish's matchMedia API Polyfill
github.com/paulirish/matchMedia.js
Paul Irish has put together a polyfill for the matchMedia API, which enables support for the matchMedia API to unsupported browsers.

Jonathan Fielding's blog *jonathanfielding.com*
This blog has many posts on handling functionality changes.

there are JavaScript libraries that will make our job easier. Not only does using a library make writing responsive JavaScript simpler, libraries often handle cross browser differences, so you don't have to.

A popular library for handling responsive JavaScript is SimpleStateManager (*netm.ag/ssm-250*). SimpleStateManager is built around the concept of states, a state being the condition of the browser at a specific width. A good example of states you may already be using is where a responsive site may have a small state for mobile devices, medium state for tablets and a large state for desktop. In this example you could use SimpleStateManager to add JavaScript specific to each of your states.

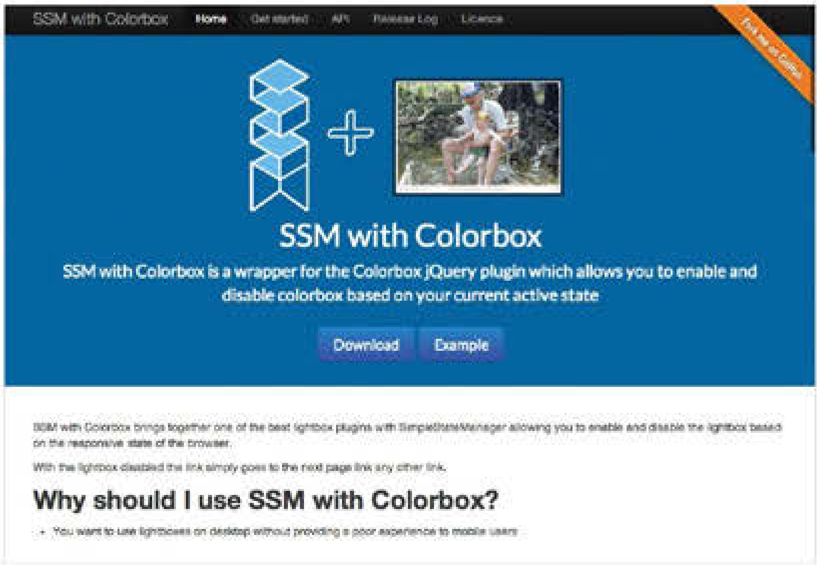
SimpleStateManager allows you to add JavaScript methods for entering, leaving and resizing responsive states based on the width of the browser. The core methodology for using SimpleStateManager is to prepare a state on the `onEnter` event, clean up the state on the `onLeave` event and use the `onResize` event for handling the resize of the state.

A popular library for handling responsive JavaScript is SimpleStateManager

There are two ways in which you can get started with SimpleStateManager, the first of which is to use Bower (*github.com/bower/bower*). Alternatively you can download the library's JS file from *www.simplestatemanager.com* and include it in your project. You can then start adding states:

```
(function(window){
  ssm.addStates([{
    id: 'mobile', maxWidth: 767, onEnter: function(){
      console.log('enter mobile');
    }
  },
  {
    id: 'desktop',  minWidth: 768,  onEnter: function(){
      console.log('enter desktop');
    }
  }]);
  ssm.ready();
}(window));
```

In this example, we add two states: mobile and desktop each with an `enter` event that logs out to the console which state we have entered. We then fire a `.ready()` method to tell SSM that we have added our states and we are ready to run any `onEnter` methods required for the current state.



SSM with Colorbox

SSM with Colorbox is a wrapper for the Colorbox jQuery plugin which allows you to enable and disable colorbox based on your current active state

SSM with Colorbox brings together one of the best lightbox plugins with SimpleStateManager allowing you to enable and disable the lightbox based on the responsive state of the browser.

With the lightbox disabled the link simply goes to the next page link any other link.

Why should I use SSM with Colorbox?

- You want to use lightboxes on desktop without providing a poor experience to mobile users

Add plugins You can add plugins to SimpleStateManager to add extra functionality

SimpleStateManager allows you to add infinite states, which can overlap one another, meaning you do not need to duplicate code between states. While it's possible to add infinite states, please bear in mind the performance implications of adding too many states.

The key advantages of using SimpleStateManager is that it makes it really simple to add states to your browser in a way that offers great performance. SimpleStateManager also provides you with the APIs to write your own plugins to extend the built in functionality. This means you can extend it to support feature detection like geolocation and touch. Unlike libraries that use the matchMedia API, it doesn't require a polyfill to work in older browsers such as Internet Explorer 7, 8 and 9.

The disadvantage of SimpleStateManager is that it's limited to only responding to changes in the width of the website. This means that it works great for the majority of responsive websites. Should your website need to carry out more complex queries you have the option to either write a plugin to extend SimpleStateManager. Alternatively, you can use a library that utilises the matchMedia API.

To summarise: the key benefit of using a library like SimpleStateManager over rolling your own solution (either with the matchMedia API or the browser resize event) is that it makes writing responsive JavaScript much simpler. This means that we can concentrate on optimising our sites to work responsively rather than spending our time having to manage the process of switching between our responsive states ourselves.

VIDEO

Watch an exclusive screencast of this tutorial created by Tuts+ Premium: *netm.ag/tut3-250*

ABOUT THE AUTHOR
GION KUNZ
w: *mindtheshift.wordpress.com*
t: @GionKunz
areas of expertise: Frontend development
q: who would play you in a movie of your life?
a: The guy who played Steve Urkel!

BROWSER SUPPORT
CHARTIST.JS

Desktop	Mobile/tablet
9	5.1
6	3
9*	-
12.1	37
5.1	32

* With matchMedia Polyfill

RWD

RESPONSIVE CHARTS WITH CHARTIST.JS

Gion Kunz shows you how to use Chartist.js to create great-looking, responsive charts in your next data visualisation project

As a responsive web developer I'm constantly seeking ways to bring standard technologies under one hood, while also trying to keep up with the W3C One Web promise (*netm.ag/oneweb-261*). Making the same information, design and interactions available on every medium – despite each coming with their own particular restrictions – may seem impossible at first. It's one of the most challenging parts of a developer's job, but I still think that this is a promise we can and should fulfil.

The problem with not displaying some content on certain media, or dismissing some interactions, is not really that it breaks a W3C contract, but that it messes with our users' expectations. That's where we pay the fine. There's nothing more annoying than discovering your favourite button on a website does not exist when you're sitting in the bus and feel like pressing it! In my experience this can even cause a user to simply choose not to view a website on a given media completely.

I think these principles should be applied to any content on the web, and this includes data visualisation. Faced with a project for a client who wanted me to implement a performance dashboard that should be accessible on a mobile and tablet, but also displayed nicely on a big widescreen television, I felt lost with the options available.

There were already tons of good charting libraries out there, but none of them actually solved the issue of generating simple and nice looking charts that also behaved responsively. After spending hours trying to tweak existing libraries to make them behave how I wanted them to, I simply decided to create my own. This marked the birth of Chartist.js (*netm.ag/chartist-261*).

GETTING UP AND RUNNING

The easiest way to started with Chartist.js is by using Bower package manager (*bower.io*). You can then install the latest version of Chartist.js by typing the following command in your shell:

```
bower install chartist --save
```

You will now have Chartist.js installed locally, and awaiting your further instruction. All you need to do is add the resources into your HTML and you're ready to draw some nice responsive charts.

```
<link rel="stylesheet" href="bower_components/chartist/libdist/chartist.min.css">
<script src="bower_components/chartist/libdist/chartist.min.js"></script>
```

Chartist.js comes without any dependencies and has a compressed size of less than 10KB. The core purpose of Chartist.js is to solve one and only one problem, which is to enable developers to draw simple, responsive charts. It does that using the power of web standards, like inline SVG in the DOM, and CSS for its appearance. With this clear separation of concerns, and leveraging standard technologies, Chartist.js is able to provide you with all you need and nothing more.

The biggest issue I'm currently noticing with a lot of libraries is that they are trying to solve too many problems, and forgetting about the fact that they are libraries and not applications.

I'm sure you've run into situations where a library customises everything but the one thing you'd really like to customise. By building upon web standards, you can easily extend the functionality of Chartist.js and scale linearly.

HACKING YOUR FIRST CHART

Now as you've included Chartist.js into your project, you can go ahead and create your first chart. The library comes with some default styles that can easily be overridden or customised with the Sass (SCSS) version.

If you'd like to go with the default styles, you can simply use the pre-built CSS class `.ct-chart` to create a chart container in your HTML.

```
<div class="ct-chart"></div>
```

That's already it for the HTML part. Of course it's up to you if you'd like to use a `<div>`, or any other element you think matches the context better.

Chartist.js tackles only one problem: to enable devs to draw simple, responsive charts

Now let's go ahead and initialise a simple line chart with two series by using the function `Line` in the global Chartist.js namespace.

```
Chartist.Line('.ct-chart', {
 labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
 series: [[0, 3, 2, 8, 9], [1, 2, 3, 5, 8]]
}, {
 width: '300px',
 height: '200px'
});
```

Congratulations, you've just created your first chart using Chartist.js! Wasn't that a piece of cake? But wait … isn't that supposed to be a responsive charting library? Of course I wouldn't recommend that you use fixed dimensions for your chart, based on pixels, right?

ASPECT RATIOS AND SCALES

Because of the nature of responsive design, it's important to understand that block content in design, including images, videos and the like need to be able to scale and adapt to the medium. In order for an element to scale, you need to rely on certain aspect ratios (4:3, 3:2, 16:9 and so on) rather than specifying a fixed width and height.

With Chartist.js you can specify those ratios directly on containers, without the need to calculate any fixed dimensions. In order to create a chart that uses the aspect ratio of a golden section, you can just add the class `.ct-golden-section` to your container where you initialise your chart.

There are also classes for other common scales that you can use instead. Just check the Chartist.js documentation online (*netm.ag/chartistdoc-261*) to get a complete list of aspect ratio classes you can use, out of the box.

```
<div class="ct-chart ct-golden-section"></div>
```

You can then just omit the fixed width and height in the options when initialising your chart.

```
Chartist.Line('.ct-chart', {
 labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
 series: [[0, 3, 2, 8, 9], [1, 2, 3, 5, 8]]
});
```

Your chart will now use 100 per cent of the available width and scale with the golden section as fixed aspect ratio.

RESPONSIVE SUGAR TOPPING

Chartist.js provides an easy way to specify configuration that should be used specifically for a given medium. It makes use of CSS media queries and `window.matchMedia` to provide a configuration override mechanism that allows you to fully re-configure your charts based on media query rules.

The following example shows a mobile-first configuration style where small screen devices will see a point on the line chart and on larger screens the points will not be drawn.

```
Chartist.Line('.ct-chart', {
 labels: ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday'],
 series: [[12, 9, 7, 8, 5]]
}, {
 showPoint: true
}, [['screen and (min-width: 640px)', { showPoint: false }]]);
```

I hope this has given you a good overview of the philosophy of Chartist.js, and that you're hopefully already thinking about a next project where you could use it.

RESOURCES

LIVE CODING

Visit the Chartist.js website to check the latest updates, and hack on some charts in the browser using the live example page. *netm.ag/livecoding-261*

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/iframeGit-262*

ABOUT THE AUTHOR

JIMMY JACOBSON

w: *wedgies.com*

t: @jimmyjacobson

job: Co-founder and CTO, Wedgies

areas of expertise: JavaScript, NodeJS, Full-stack development, scalability

q: how do you deal with stressful situations?

a: Take a walk! When a situation or event is getting stressful, I like to take a quick walk around the block

RWD

ADD 3RD PARTY CONTENT TO RESPONSIVE SITES

Jimmy Jacobson explains how to use JavaScript, iframe and RWD techniques to embed third party web apps in your responsive site

Ah, the iframe tag. Before tables and CSS, it was all a web designer (or, as it was back then, a webmaster) had to control the layout of a page. People did crazy things with iframes, including trying to mimic the sidebar of Windows apps. As HTML evolved to support tables and then CSS, the iframe fell into disuse and even disdain. The only acceptable use of the iframe was to display ads.

Good news – the iframe is back! Actually, it never really went away. You just haven't been noticing it any more. Services like Disqus, Vine, CodePen, Tumblr, Wedgies and SoundCloud all use iframes to enable social media networks and third party websites to embed fully featured web apps. These apps offer users the ability to interact with content such as videos, user authentication and even social features – so please don't call them 'widgets'.

- Disqus is a commenting platform that turns any page into a forum thread with comments
- Twitter uses iframes generated via JavaScript to enable users to embed Tweets inside any web page
- Tumblr uses iframes to display videos and other content, as well as to prevent malicious JavaScript from being run inside its dashboard
- CodePen makes use of iframes to show real time previews of code being written by its users, and to embed those previews on third party websites

ADAPTING AND ADJUSTING

Living in the 21st century, content has to work on an array of devices. And that is true for these embedded web apps. Websites often use different layouts for mobile and desktop rendering. If the embedded web app does not adjust its display for different devices, it will not fit inside the mobile layout, ultimately causing the website owner or social network to abandon it.

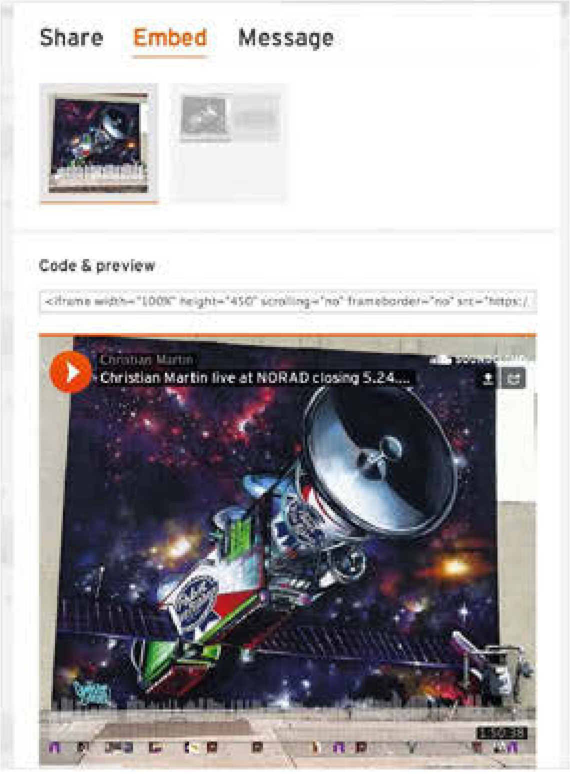
> These apps include the ability to interact with content – please don't call them 'widgets'

The solution to this is the same one that web designers and developers use for websites on mobile devices. Responsive design techniques, including using a fluid grid, flexible images and CSS media queries, work just as well when an HTML document is rendered inside an iframe as they do when the document is the root of the window.

There is one glaring issue that keeps this from being a perfect solution. It is impossible for an iframe to have a height specified by a percentage,

RESOURCE

3rd Party JavaScript by Ben Vinegar and Anton Kovalyov covers writing JavaScript to be used by third party websites. Of special interest will be the sections on writing your own library and cross-domain message passing. *netm.ag/3rd-262*



```
<iframe width="100%" height="450" scrolling="no" frameborder="no" src="https:/
```

Set height The UI for embedding a SoundCloud shows you can't set the height of an iframe to 100 per cent via its attributes

and for an iframe to detect the height of the content being rendered inside it. In most cases, when an explicit height in pixels can't be detected for an iframe, browsers default to 150 pixels.

FINDING A FIX

At Wedgies, our solution to this problem comes from the excellent book *Third Party JavaScript* by Ben Vinegar and Anton Kovalyov (*netm.ag/thirdparty-262*). Using JavaScript, an HTML document embedded in an iframe can pass a message to the parent window using cross-domain messaging.

The flow is pretty simple, but there are some specific things that have to be done in order to conform to the security policies of various browsers. Internet Explorer, in particular, has some very strange requirements (see the GitHub repo at *netm.ag/iframeGit-262* for more on this).

1 Register an event handler in the embedded document for resize events
2 The embedded document posts a message to the parent window that contains its own height. This height is detected via the height attribute of the embedded document in the resize event handler
3 The parent window listens for the message and resizes the iframe element whenever the message is detected, using the data in the message

▶

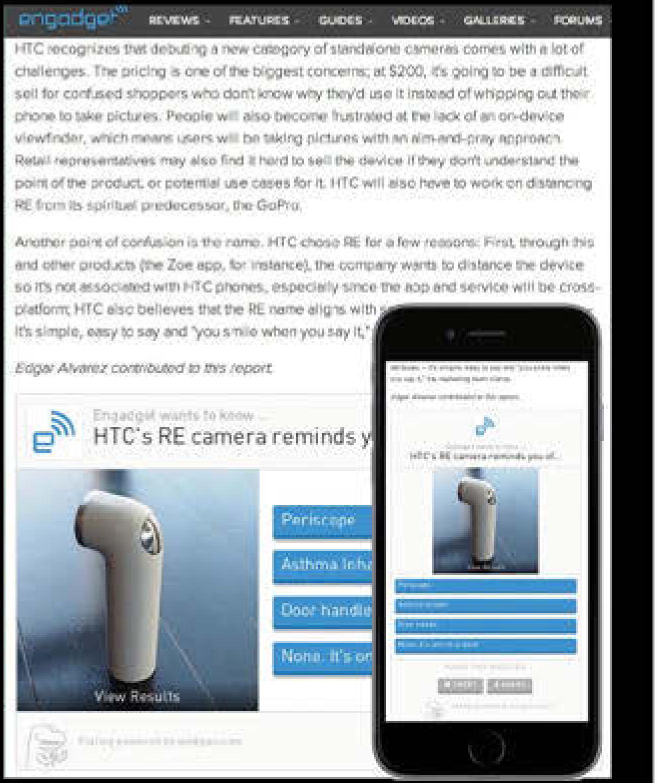
CASE STUDY

EMBEDDING POLLS

Wedgies (*wedgies.com*) provides a platform for creating and voting on polls. A lot of Wedgies' users choose to embed these polls on their own websites. This is how the development team at Wedgies learned how difficult it could be for a user to embed third party content into a responsive layout.

Wedgies experimented with a few different ways to make this process easier on the user, including allowing the user themselves to set a height and width on the content. But in the end we found the best solution, as ever, was the one that removed the worry from the end user. Just make it work.

The technique we now use is achieved through a combination of JavaScript that is loaded into the user's website, and responsive design in the content provided to the iframe. Below you can see two examples of one poll embedded into both a desktop layout and a mobile layout. The content has adapted responsively and the iframe has been resized automatically. In both examples, the code provided to the end user and the code served into the iframe are identical.



HTC recognizes that debuting a new category of standalone cameras comes with a lot of challenges. The pricing is one of the biggest concerns; at $200, it's going to be a difficult sell for confused shoppers who don't know why they'd use it instead of whipping out their phone to take pictures. People will also become frustrated at the lack of an on-device viewfinder, which means users will be taking pictures with an aim-and-pray approach. Retail representatives may also find it hard to sell the device if they don't understand the point of the product, or potential use cases for it. HTC will also have to work on distancing RE from its spiritual predecessor, the GoPro.

Another point of confusion is the name. HTC chose RE for a few reasons: First, through this and other products (the Zoe app, for instance), the company wants to distance the device so it's not associated with HTC phones, especially since the app and service will be cross-platform; HTC also believes that the RE name aligns with
It's simple, easy to say and "you smile when you say it,"

Edgar Alvarez contributed to this report.

Engadget wants to know ...
HTC's RE camera reminds y

Changing shape Shown here is a Wedgies poll embedded into a page rendered on a large screen. Next to it is the same poll embedded into the same page, but rendered in a mobile screen size

Take a look at the following code, which goes in the webpage that is embedded via iframe. Use of jQuery is assumed in this example.

```
function postMessage() {
var msg = $('body').height();
window.parent.postMessage(msg, '*');
}
```

Event handler for window resize:

```
$(window).resize(postMessage);
```

Event handler for page load:

```
$().ready(postMessage);
```

This code is loaded by the parent webpage. It listens for a message and sets the height of the element with `id = id-of-iframe` to the value in the message. This is pretty simple code.

In a production environment there are many things posting messages, so be careful about filtering for the specific message from your app.

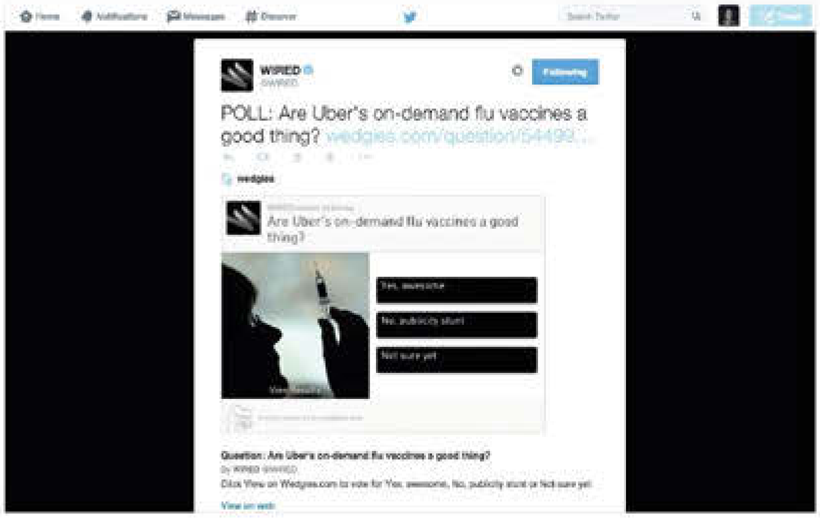
```
function msgReceiver(ev){

var msg = ev.data;

var elem = document.getElementById('id-of-iframe');
if (elem) {
elem.style.height = (parseInt(msg)) + "px";
}
}
```

This technique requires the developer of the embedded web app to have control over JavaScript on the parent page. The best way to do this is to give them a combination of a JS library and a snippet of HTML containing a unique ID. When run, the JS library detects these snippets and expands them into the code needed to render the iframe containing the correct content. The library also contains the message listener and code to resize the iframe when a message is received.

This technique allows the iframe to be resized along with the page, and for the document embedded in the iframe to respond to the resize events responsively. Now when a user pulls up the web page on a mobile device, the embedded web app responds to the width and height correctly, instead of being rendered in a fixed height iframe, cutting off the content in the mobile layout.

Top Disqus is deployed to third party websites via JS, and generates an iframe to load the web app

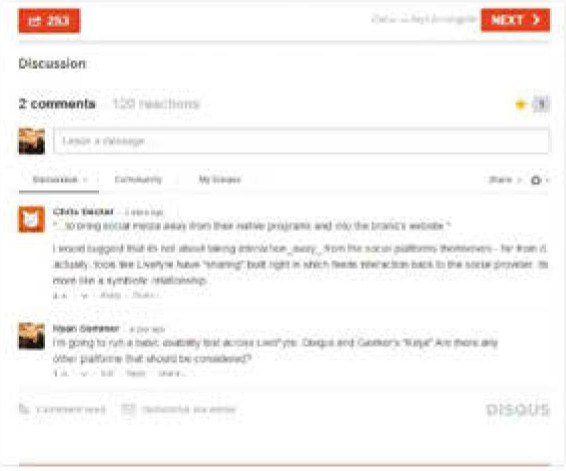
Below Wedgies uses PhantomJS to render the embedded web app view of a poll, which can be used as content inside Facebook and Twitter posts



POLL: Are Uber's on-demand flu vaccines a good thing? wedgies.com/question/54499…

Are Uber's on-demand flu vaccines a good thing?

Question : Are Uber's on-demand flu vaccines a good thing?
by WIRED @WIRED
Click View on Wedgies.com to vote for Yes, awesome, No, publicity stunt or Not sure yet.



Chris Decker
"...to bring social media away from their native programs and into the brand's website."
I would suggest that it's not about taking interaction _away_ from the social platforms themselves - far from it, actually. tools like Livefyre have "sharing" built right in which feeds interaction back to the social provider. Its more like a symbiotic relationship.

Ryan Gemmer
I'm going to run a basic usability test across Livefyre, Disqus and Gawker's "Kinja" Are there any other platforms that should be considered?

BENEFITS AND BONUSES

Leveraging responsive design inside a web app intended to be embedded on third party websites has other bonuses as well:

- By providing an oEmbed endpoint, the web app can be discovered by services like Embedly (*embed.ly*) that aggregate embeddable services, exposing the app to more potential users
- Building plugins on platforms like WordPress is much easier
- PhantomJS can be used to render a screenshot of the embedded web app and used as content for Twitter or Facebook posts
- Content is ready to be used inside a mobile app via Web View, without any modifications

RWD techniques aren't just for mobile. iframes trigger the same events and constraints a document needs to render itself responsively. A web app deployed to an iframe and distributed across websites will have broader adoption if it works across any screen size or mobile device.

TOUCH

POINTER EVENTS

Tomomi Imura discusses touch-first development using pointer events

Event types including touchstart, touchend and touchmove are fired only on browsers that support touch events on touch-enabled devices. When you build an app for multiple environments, you don't want to discriminate against your users by which device and user-input they're using. To make both touch and mouse user input methods work, you need to support touch and mouse events together.

You probably wish there was a single set of events to handle both the iOS touch event model and the standard mouse events. Well, Microsoft has been working on MSPointers to support multi user-inputs, and it has now reached W3C Proposed Recommendation stage. This event model is called Pointer Events and is designed to offer adaptive input-handling, allowing hardware-agnostic pointer input from devices like a mouse, pen or touch. It has been partially implemented in IE 11, and Mozilla has accepted the recent implementation for Firefox.

The example below gets the x and y pointer coordinates relative to the upper left edge of the browser window, using the pointer events. You'll notice the code looks very similar to mouse events.

```
window.addEventListener('pointerdown', function(e) {
   console.log('pointerdown', e.clientX, e.clientY);
}, false);
```

The event is fired when any pointing makes contact with the screen, for instance, a finger touching down on the screen. This is also a compatible mouse event. After firing a pointer event, the mouse events for the primary contact will be fired as well.

To support IE10, you must use the prefixed version, so pointerdown, pointermove and pointerup should be written as MSPointerDown, MSPointerMove and MSPointerUP respectively. While supporting IE11, you can drop the prefix. The feature detection for the pointer events can be written:

```
if(window.PointerEvent) {
  // Pointer events supported!
}
```

Also, you need to enforce the touch behaviours by disabling mobile browsers' default behaviours, such as zoom and scroll with CSS:

```
touch-action: none;
```

The only available browsers that support the Pointer Events are IE10 and above, while Firefox is catching up, so you need to include the touch and mouse events until the feature id is available.

```
var isTouchSupported = 'ontouchstart' in window;
var isPointerSupported = (window.PointerEvents || window.
MSPointerEvents);
var downEvent = isTouchSupported ? 'touchstart' :
(isPointerSupported ? 'MSPointerDown' : 'mousedown');

el.addEventListener(downEvent, function(e) {
  var x = isTouchSupported ? e.targetTouches[0].clientX :
e.clientX;
  var y = isTouchSupported ? e.targetTouches[0].clientY :
e.clientY;
}, false);
```

You can see a similar interactive demo at CodePen (*netm.ag/pointer-253*).

PROFILE

Tomomi (*girliemac.com*) is an open web advocate and frontend engineer, who writes about HTML5, CSS, JS, UX and mobile. She works as a developer evangelist at PubNub in San Francisco

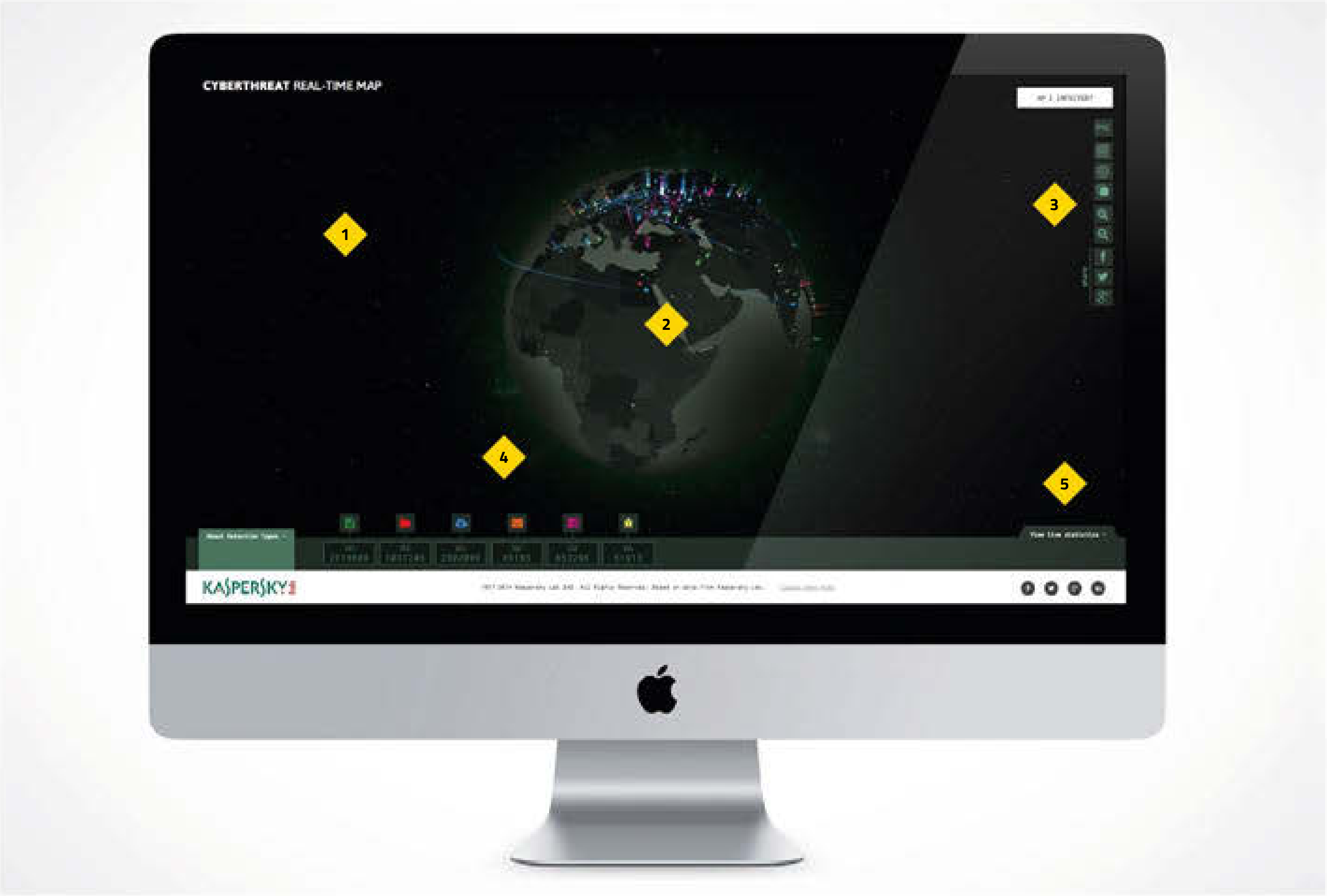
WEBGL

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

HOW WE BUILT

KASPERSKY CYBERTHREAT

How Minivegas created a dramatic and dynamic Cyberthreat map to visualise the attacks dealt with by Kaspersky Lab



BRIEF

Every day, Kaspersky Lab products detect millions of cyber attacks around the globe and stream that data back to headquarters. The team wanted to visualise this data and bring it to life in a compelling, dynamic interactive experience.

CLOSE UP

(**1**) The heart of the experience is an interactive globe that rotates in space and gives visitors an overview of what's happening. (**2**) Selecting a country zooms the viewer in. A pop-up reveals that country's 'infection' rank and the number of detections made by the different Kaspersky Lab products. (**3**) In the toolbar, the user can change languages, switch between globe and flat map views, alternate between different colour versions and zoom in or out. (**4**) The detection counter icons represent the different Kaspersky Lab products. Each counter tallies the number of cyber-threat detections made in the past 24 hours. Users can also use the icons to isolate particular types of threat on the map. (**5**) A 'View live stats' tab pops open a drawer to display how many detections are being made by each product in real time, as well as a list of the top five most infected countries.

ANDREW WATSON

Andrew is creative director at Minivegas and co-founder of Facebook phenomenon Crapnet
w: *minivegas.net*
t: @watsonamac

TOBIN NAGEOTTE

Tobin is interactive producer at Minivegas and co-hosts the extremely highbrow thirtysevenclick podcast
w: *minivegas.net*
t: @37click

BRIAN BOURKE

Brian is executive producer at Minivegas and former frontman for almost nearly famous Irish band Hoju
w: *minivegas.net*
t: @brianhoju

Kaspersky Lab sees literally millions of attempts to steal consumers' private data happening every day – in 2013 it fixed more than five billion cyber-attacks on users' computers and mobile devices all over the world. The security specialist brought in Minivegas to help show the world what it sees, in the form of a dynamic visualisation. We spoke to the team at Minivegas to find out how they rose to the challenge.

Why don't you introduce yourself and your agency?
BB: We are Minivegas, we work with agencies and brands to deliver cutting-edge work. In essence, Minivegas is an integrated production company working across the four disciplines of digital, live action, animation and VFX, and experiential. It is ingrained within our company culture to work with the latest technologies, which is why we get to create projects such as the Kaspersky Cyberthreat Map.

What is the Kaspersky Cyberthreat Map, and what is it intended to convey?
BB: The website visualises all virus attacks detected by Kaspersky Lab software around the globe, in real time. It displays what countries are being hit, the different types of attack and, in some cases, the origin of the attacks. The result is a really spellbinding, dynamic representation of a real life problem that normally doesn't get exposure like this.

What were client briefings like?
BB: The guys at Kaspersky Lab were really open and clear. When they first approached us, they outlined core technical requirements and some visual references – they had a clear vision of what they wanted the site to show and were very open for us to suggest how to show it.

The site looks amazing. Did the client ask for something very dramatic?
BB: Thanks! Kaspersky Lab definitely wanted something dramatic. The subject matter is a bit sinister and we really wanted that to come across in the design style. Some of the references discussed were the popular computer game DEFCON and the Cold War flick *WarGames*, and the final design is based on computer graphic war games style from the 1980s. Of the six products Kaspersky was mining data from, two showed the source of the threat as well as the destination. This enabled us to incorporate meaningful trajectories as well as static detections, and make the map really come alive.

At the heart of this project is the clear visualisation of data so the user instantly understands what is being shown. Once that dramatic first impression has taken place, the idea is that people take the next step and start to interact with the globe and find out more.

How did the design evolve?
BB: We started with moodboards and style frames for the map. Once we had agreed

▶

★TIMELINE

Mapping the Cyberthreat, from raw data to real time visualisation

OCT 2013
Kasperksy Lab approaches Minivegas with the idea of mapping its data into some kind of interactive web experience.

NOV 2013
Planning and prototyping kicks off the visual concepting phase, including regular reviews with the team at Kaspersky Lab.

DEC 2013
Minivegas locks its visual direction for the project and starts WebGL development.

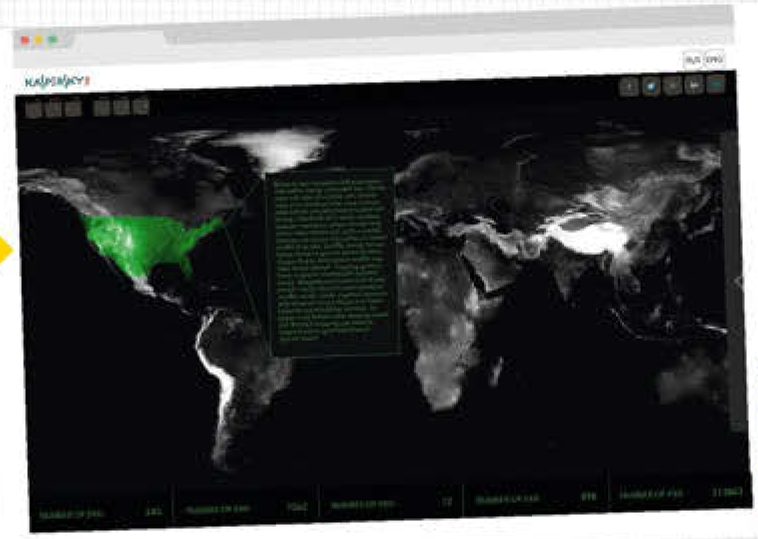
DEC 2013
In parallel to the WebGL development, the team begins wireframing the user experience and exploring the UI design.

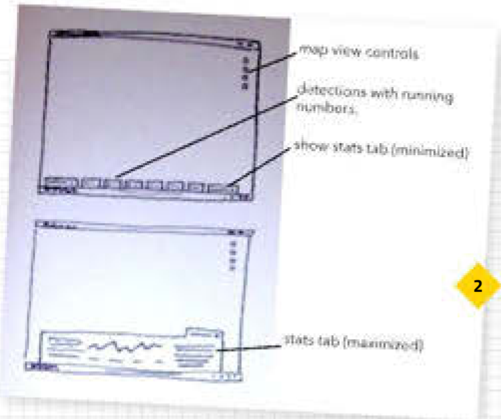
EARLY JAN 2014
With the user interface approved, HTML development could begin in the new year.

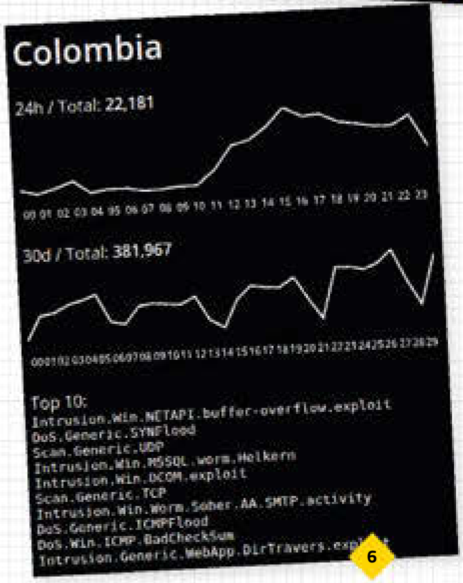
LATE JAN 2014
At an intermediary stage of development, the team tests the first integration of the WebGL with the HTML.

EARLY FEB 2014
The final stage of production is QA & browser testing along with performance optimisation.

LATE FEB 2014
The final approvals are given by the client and Minivegas pushes the site live.







Colombia

24h / Total: 22,181

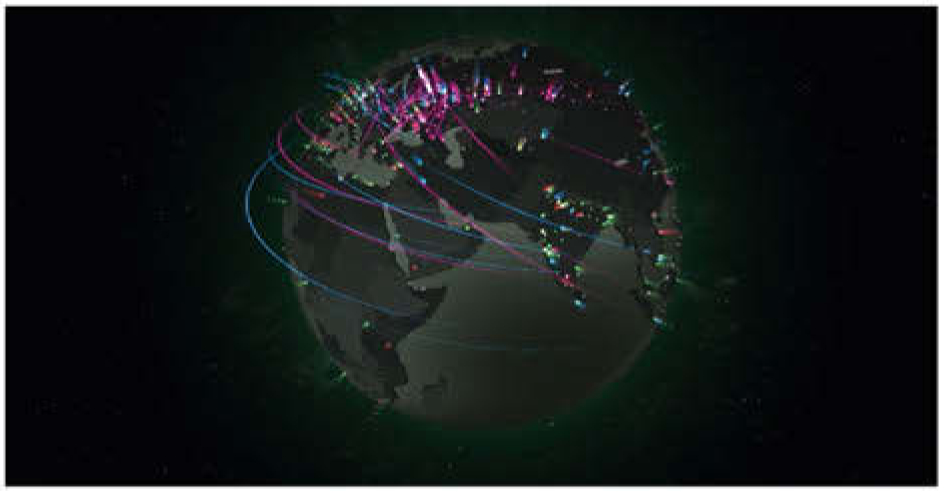
30d / Total: 381,967

```
Top 10:
Intrusion.Win.NETAPI.buffer-overflow.exploit
DoS.Generic.SYNFlood
Scan.Generic.UDP
Intrusion.Win.MSSQL.worm.Helkern
Intrusion.Win.DCOM.exploit
Scan.Generic.TCP
Intrusion.Win.Worm.Sober.AA.SMTP.activity
DoS.Generic.ICMPFlood
DoS.Win.ICMP.BadCheckSum
Intrusion.Generic.WebApp.DirTravers.exploit
```

EVOLUTION

The concepts and ideas that were explored as Minivegas developed its idea

(1) An early layout. With a one-page site there wasn't much wireframing to be done up-front – it was more about distributing information clearly. **(2)** We did a lot of rough scribbles, snapped them in Photo Booth and got feedback from the team. **(3)** A lot of time went into style exploration and research. This moodboard explores mapping examples. **(4)** Another moodboard, this time looking at explosions. **(5)** At one point, we explored the combination of realistic map renders with the digital detection overlays. **(6)** A later development. We liked this kind of raw data feel, but ultimately the site needed to be clear and understandable.



A global view Taking a wide view, it is fascinating to watch just how much is going on around the world

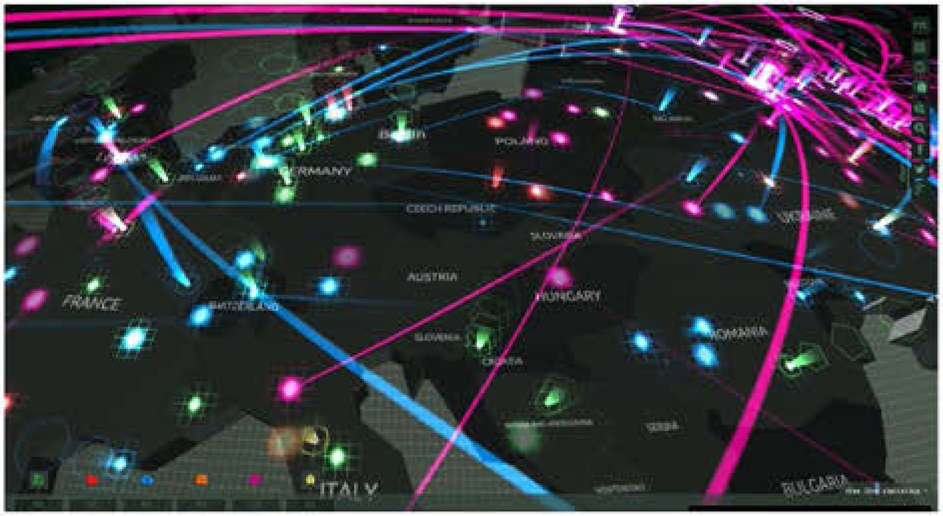
a look for those elements, we explored how the detections could come to life and worked on the user interface. While the map was translated into code, we designed the UI elements that would sit on top of the dynamic WebGL map.

Once the visual design was locked we built out the HTML5 interface and fine-tuned the globe visuals. For all real-time data visualisations in WebGL we get the functionality locked early, and this allows us to work on the final piece of the puzzle: finalising the animated elements, to get the look just right.

Which technologies did you use?
BB: We built the globe in WebGL, which is an incredibly efficient 3D rendering API for modern browsers. We've been working in WebGL for the last three years, as we always believed it offered a lot of versatility for data visualisation and gameplay in browsers – in the future there will be a lot more sites that use it. The rest of the user interface was built with HTML5, JavaScript and CSS. On the server side, we take raw data feeds from Kaspersky Labs and compress them into a binary form that's quick for the client to decode and process. These are just fetched via standard AJAX, rather than WebSockets or anything fancy.

The site is very smooth. How did you maximise performance?
BB: The speed of JavaScript today is really impressive – you can do a lot at 60fps without optimising that much. A generally good practice for WebGL code is to not allocate objects in the animation loop. We drew our scene in batches to minimise state changes – so first the missile trails, then missile impacts, impact shapes and so on – rather than drawing things missile by missile. In addition to this, we kept expensive per-frame calculations such as missile arcs and dynamic map projections in the shaders where they belong.

How much data is used to power the visualisation?
BB: The core data supplied by Kaspersky Lab defined specific types of attack that its products detect, where they originate and where they hit, along with when these detections happen. However, we didn't fully realise the sheer volume of detections to be handled until we got our first bit of sample data from Kaspersky. Quantity aside, there is a limit to the data Kaspersky Lab exposed to us for both technical and security reasons. The locations of the attacks are by country, and we then spread these across the countries based on population density.



Battle ground As the viewer zooms in, they are plunged into the maelstrom of malicious cyber activity

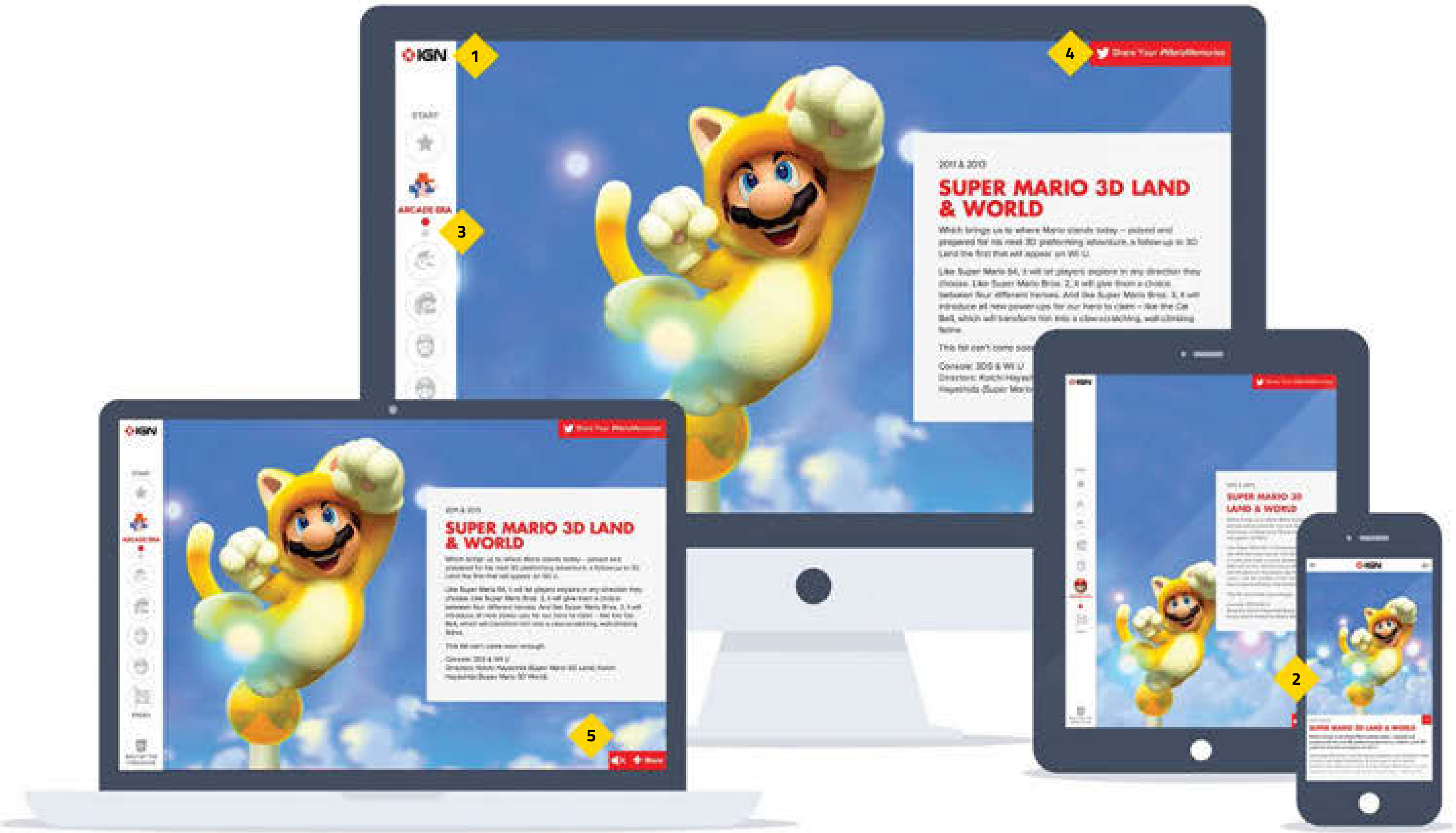
How did you go about prototyping and communicating ideas to the client?
BB: The key was to keep the process structured and ordered so that each aspect could come together in order and the project stakeholders weren't continuously having to go back to the drawing board. Prototyping plays a crucial role at the beginning of most of our projects. We spend a week getting all of the pieces in place and have a conversation with the client around that. This prototype is accompanied by the design documentation, including sketches, wireframes and an initial look and feel. Then, with as much agreed as possible, we begin to make.

What's reaction from the web community been like?
BB: We have had some incredible feedback, as well as accolades from Awwwards and the FWA including the Adobe Cutting Edge award, which was a great honour to receive.

HOW WE BUILT

THE MUSEUM OF MARIO

Mario is one of the most iconic games characters. Kyle Simpson of **HTML5 Hub** talks about creating an online Museum of Mario



BRIEF

Intel wanted to inspire HTML5 developers and drive conversation around innovation and technology. This led to Intel's HTML5 Hub teaming up with IGN to create an interactive experience exploring the many eras of Mario.

CLOSE UP

We teamed up with IGN to take one of their upcoming articles and transform it into an interactive experience. (**1**) The site lived on a sub-domain of IGN, further adding credibility and driving traffic. (**2**) The entire site was built to be responsive across all devices. This was especially challenging given the modern web technologies and interactivity we built into the experience. (**3**) Rather than have a standard vertical scrolling experience, we incorporated what we called 'paged' scrolling into the site. This created a unique navigation experience that also transferred well to all mobile devices. (**4**) Users were encouraged to share their 'Mario Memories' on Twitter, creating a social component in the site that led to a broader exposure. (**5**) Sound was an important aspect of the site that brought a sense of nostalgia back to each game screen.

KYLE SIMPSON

Kyle is an open web evangelist, author, workshop trainer, tech speaker, and avid OSS community member

w: *html5hub.com*

t: @getify

"It's-a me, Mario!" Everybody loves Shigeru Miyamoto's Italian plumber, so when they saw IGN's Museum Of Mario (*mario.ign.com*), the creatives at HTML5 Hub were captivated. The site is a authentic and interactive retrospective exploring the character's evolution, beginning with a Donkey Kong cabinet and tracking the character's growth right up to the Wii U. Here, Kyle Simpson – one of the creators behind the site – explains how it came about ...

Hello! Why don't you go ahead and introduce yourself?

KS: I'm Kyle Simpson, the lead developer of the Museum of Mario site project and a contributor to HTML5 Hub.

And the Mario project, how did that come about?

KS: We brainstormed about a project that would push the boundaries of HTML5 and JS in browsers and devices, connect with visitors' memories, and would also be super fun. We all like games, and Mario is the most iconic game character of the last 30-plus years, so it was a no-brainer. He deserved a fitting tribute, and that's what we set out to do. We approached IGN to partner with them on it, and they were instantly hooked, too!

People love Mario. Why do you think even grown adults have such an attachment to the character?

KS: Most grown adults actually grew up playing Mario across dozens of different games on a wide array of game systems, from consoles to handhelds. Nintendo kept pushing Mario into newer, more immersive experiences, and we all tagged along for the ride. Some of us can't remember gaming without Mario. He's a part of our collective memories. Mario represents the best ideals in gaming. He exemplifies good, clean fun, but still with plenty of challenge and complexity. He transports us into his world, where we can escape, rather than making us more aware of the complications of our own world.

When dealing with Mario and his friends, are there any Intellectual Property rules you need to adhere to?

KS: The code for Mario is totally open source (MIT licensed), and all of the assets we used were obtained through various means in the broader web arena. That having been said, the assets and trademarks are still entirely Nintendo's. I don't think our project necessarily sets any precedent for others' attempts at piggybacking on Mario and friends, but there have obviously been lots of other sites and game remakes, so it doesn't seem like the Koopa Troopas are coming to get you!

How did the site's defining interactions begin and evolve?

KS: We started out trying to recreate parts of the games we wanted to highlight. We eventually settled on highlighting the most iconic and memorable experience

▶

★TIMELINE

The evolution of the Museum of Mario site, from initial meeting to going open source

JUNE 2013

First meeting with IGN's editorial team to discuss content ideas. They settle on creating something around the history of Mario.

JULY 2013

IGN sends over a rough draft of the copy and the brainstorming process begins.

EARLY AUGUST 2013

Early designs emerge as the team begins to identify a 'hero' composition for each game.

LATE AUGUST 2013

Development of site architecture begins while first round of designs is still being finalised.

EARLY SEPTEMBER 2013

Development kicks into full gear as we begin creating the interactive elements for each game screen, constantly iterating between design and development.

LATE SEPTEMBER 2013

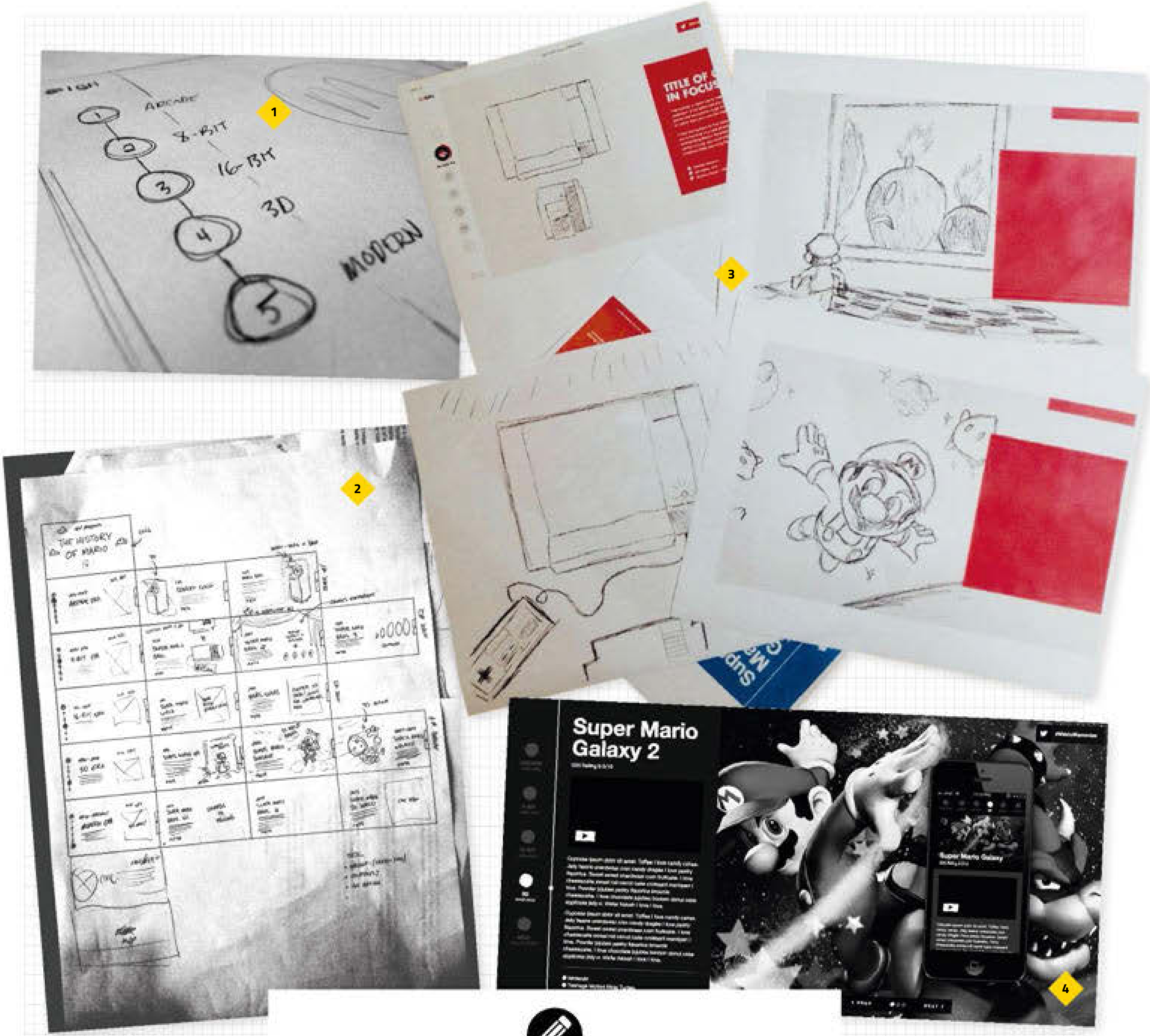
QA begins. This is a tedious process with so many interactive elements on multiple devices.

EARLY OCTOBER 2013

Project launch: the site goes live on IGN with a link from its homepage.

LATE OCTOBER 2013

Project code is open sourced on GitHub.



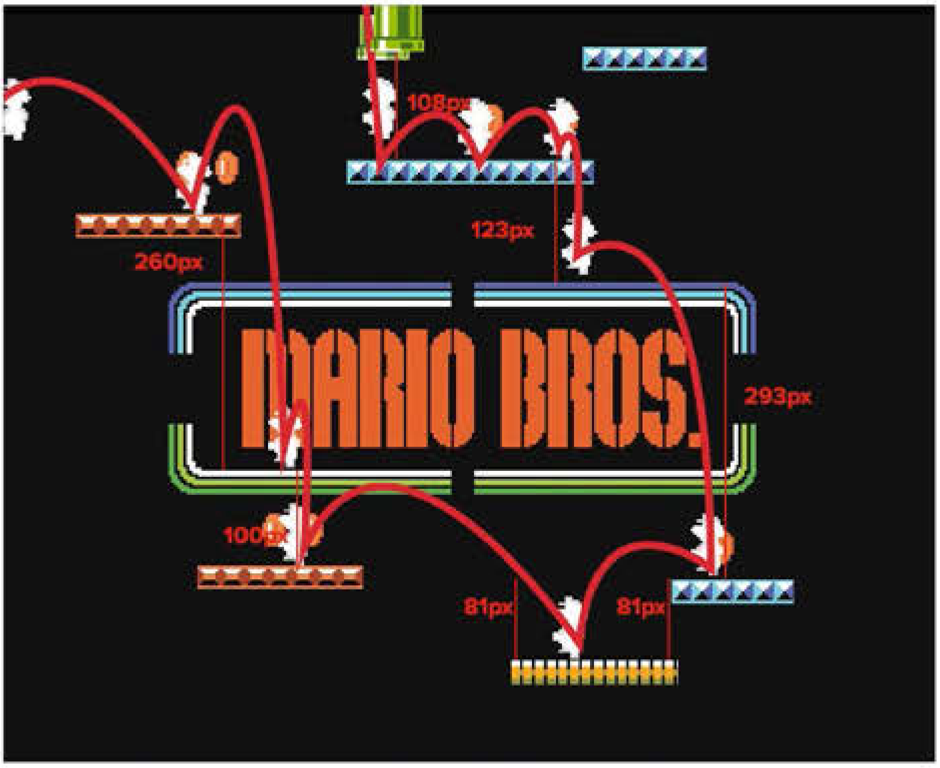
EVOLUTION

Kyle Simpson explains how the concept and design of the Museum of Mario project evolved

(1) We started with a rough layout, dividing Mario's most notable titles into different eras. **(2)** Next we began designing the global architecture and navigation of the experience. **(3)** From there, our team identified what we considered to be the 'hero' composition of each game screen. **(4)** As our developers began prototyping the architecture and navigation of the site, the rest of our design team hit the drawing board to start refining each visual concept.

from each game, and tried to weave in at least some interactivity to each one. We also wanted this to feel like entering the world of a game. Originally, you would have navigated in all four directions, just like on a game map, but complexities of devices landed us at the 'paged-scrolling' vertical experience.

I'm assuming you prototyped elements individually and then pieced the whole site together?
KS: Initially, various pieces were quite independent. Developers on the team each took one or a few of the pages and worked on them, while I (and others) worked on the overall site architecture. But the complexities of delivering a consistent experience cross-device required pretty early on in each page's life that it be integrated with the site to make sure interactions made sense and didn't interfere with the site and navigation. It was more of a holistic build and design.

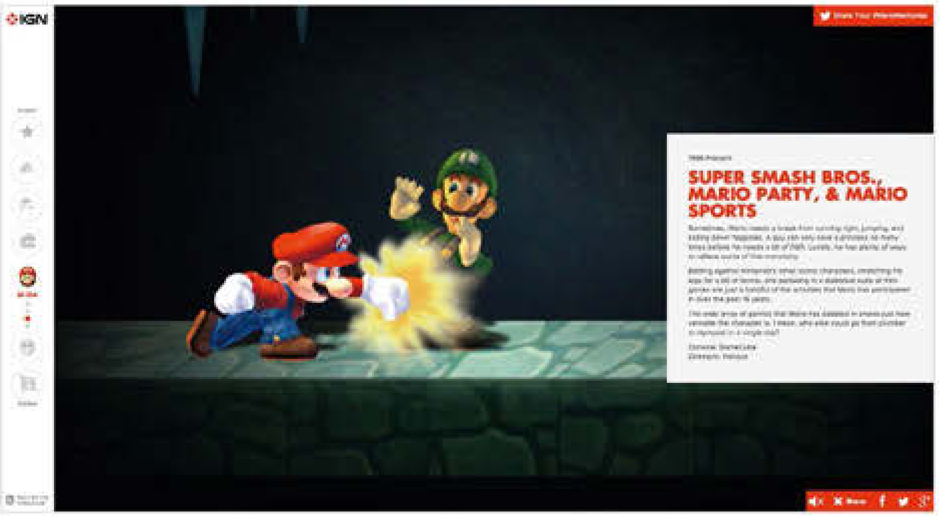


Animating Mario The team's designers worked very collaboratively with the developers to not only prototype but continually iterate on each game's concept and perfect the final product

Looking at the technologies you employed under the hood, which were most important?
KS: With literally millions of visits so far, a simple custom Node.js server backend is more than capable of handling the load without even breaking a sweat. Along with Node.js, we also heavily use Grunt and various plugins to take care of all our build and deploy tasks. Frontend wise, we relied heavily on responsive design (CSS media-queries with breakpoints), and tools like jQuery (and various plugins) to help us out with the heavy lifting. LABjs (my script loader) does all the script resource loading, and grips (my templating engine) builds all the markup (both server-side in Node, and client-side). We heavily use Canvas and three.js for some of the WebGL stuff (like the Donkey Kong arcade). Adobe Edge was essential in the initial builds of many of the sophisticated animations.

Which was the most difficult to build?
KS: The WebGL Donkey Kong arcade was tougher than most because it involved a new technology (to us) and we had to get the video positioned at just the right angle to complete the effect of the arcade screen. Mostly, though, the toughest part of this project was creating a consistent game experience in all devices. It took lots of trade-offs and compromises to get it right. We were strongly committed to making mobile a first-class experience, rather than some sites which severely limit what you get outside of desktop land. IGN has a huge mobile market share, and the web is increasingly non-desktop, so our focus had to be much broader.

One last game on one last console. What would you choose?
KS: Contra, original Nintendo. Still the most infuriating but most engaging game I ever played.



SUPER SMASH BROS., MARIO PARTY, & MARIO SPORTS

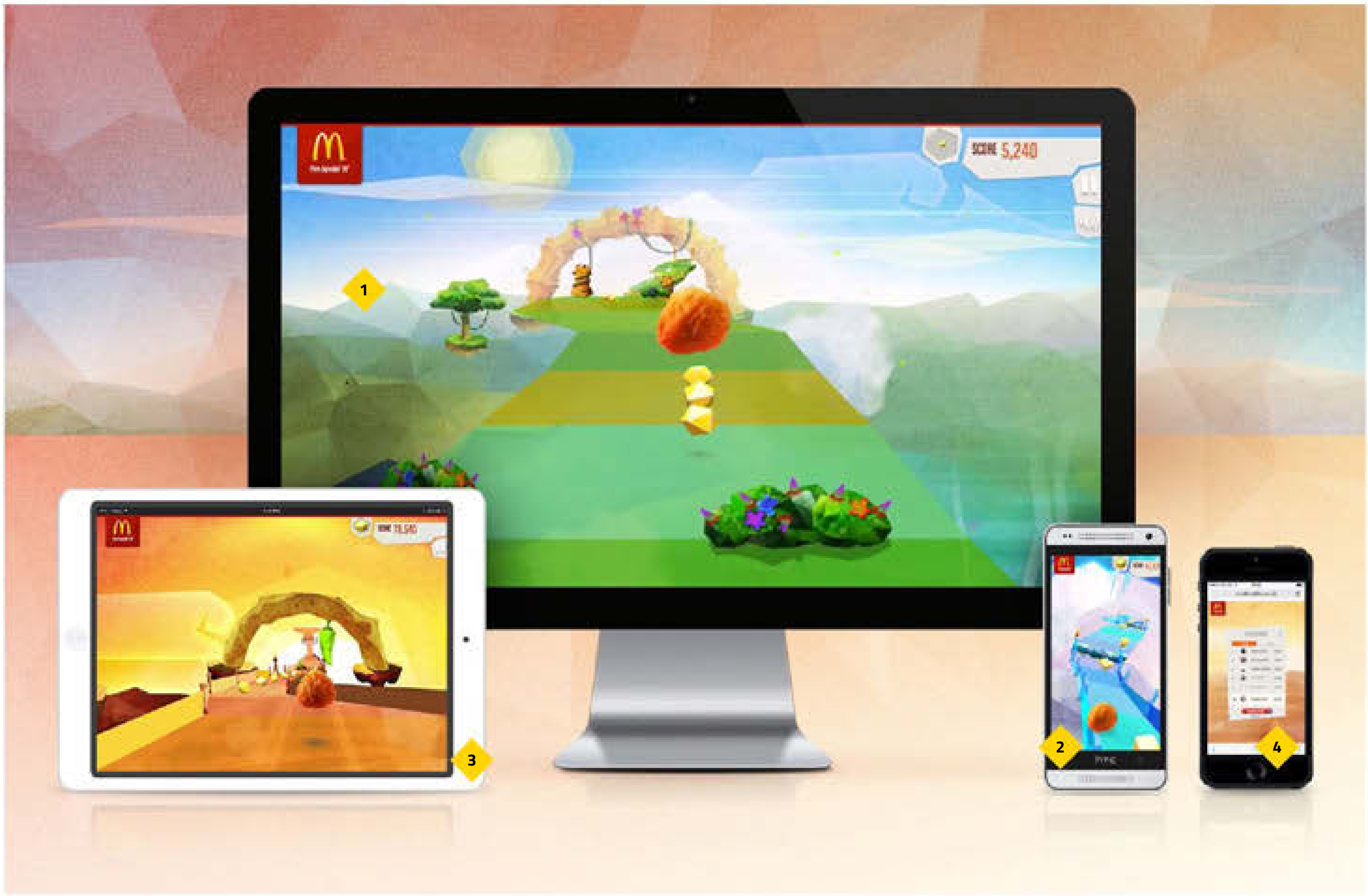
[illegible]

Finishing touches The team implemented sound effects and theme songs and polished the final product

HOW WE BUILT

SUPER SPICE DASH

Razorfish and Goodboy explain how they created the "pedal-to-the-metal-bleeding-edge-action-fest" 3D game to promote Spicy McBites



BRIEF

McDonald's Spicy McBites. Small in size, but big in spice. McDonald's asked Razorfish and Goodboy to step up to the spice challenge and position the duality of its small product with an impactful flavour in an engaging way.

CLOSE UP

(**1**) Super Spice Dash is an immersive fast-paced game. The game is an 'endless runner', using a chase-cam from a third-person perspective. The player must go as far as possible in this never-ending adventure. (**2**) The big 'wow' factor is that we've created an extremely visual, visceral 3D game in HTML5 that works on almost any device. (**3**) The game has different worlds to play: The Treetop Topple, a lush and green forest; Snowy Peak Mountain, icy and scary; and Sand Storm Mirage, a hot and rocky world full of traps. (**4**) The players are able to login with Facebook and challenge their friends. They'll see a special scoreboard made for them and their friends.

CYRIL LOUIS

Cyril is creative director on the McDonald's account at Razorfish
w: *www.razorfish.com*
t: @Millice_

JOHN DENTON

John is a creative partner and the uptight aesthetic at Goodboy
w: *www.goodboydigital.com*
t: @juannacho

MAT GROVES

A bedroom coder turned pro, Mat is a technical partner at Goodboy
w: *www.goodboydigital.com*
t: @Doormat23

I can't believe it's not Flash. That's likely to be the reaction when you sit down and feast your eyeballs on Super Spice Dash (*mcbites.sh75.net*). It's a fixed perspective racing game, created to market McDonald's Spicy McBites. The bright, brash and blisteringly fast site uses web standards to create a sensation of speed that is truly remarkable. We sit down with London agencies Razorfish and Goodboy, and discover what makes them and the site tick (so quickly).

Hello! Thanks for making time to speak to us. Why don't you introduce Razorfish and Goodboy?
CL: Razorfish London is a full-service digital agency, founded in London in 1995 with the mission of bringing technology, creativity and media together. We're now 280 people strong and experts at creative, design, social media, digital media, analytics, technology, innovation, service operations and user experience.
GB: Goodboy are new kids on the block in agency terms, although as individuals we've been in the industry for… ever. We're the 'I can't believe it's not Flash' agency, founded to make rich, entertaining content that reaches everyone, everywhere. We take a tech-first approach to creative meaning that aesthetics and performance are intrinsically linked.

And the McDonald's Super Spice Dash, can you describe the project?
CL: Super Spice Dash is an evolution of the McBites launch and a big step up. The need to reach the burgeoning iOS market meant if we wanted to keep everything 'in-browser', we'd need to start from scratch.
GB: With this in mind, we started looking into possible games that might fit with the new product. We ended up with two key approaches. One was a more simple 'legacy device' friendly 'pitch and put' type of mini-golf game. The other was the pedal-to-the-metal-bleeding-edge-action-fest. We weren't 100 per cent certain how far we'd be able to take the second option and keep everything looking and playing great. So with a mountain to climb, we all settled on option two.

How did you land the job and what was the client brief?
CL: Razorfish is the digital agency of record for McDonald's UK and responsible for most of the digital campaigns. We asked our friends at Goodboy to partner after the initial creative idea.

What is McDonald's like as a client?
CL: They're very smart and effective and commission a great number of campaigns every year. Super Spice Dash was one of them, used to launch the spicy version of the popular chicken McBites.

How did you prototype the project and communicate your ideas?
GB: Once we had settled on the loosest of game styles, we went directly to starting on proof of concept as, more than ever, we needed to prove the concept could actually work at all!

▶

★TIMELINE

Razorfish and Goodboy's McBites adventures

19 AUGUST 2013
The project starts with the question: what adventures did the McBites go on to come back so hot and spicy?

30 AUGUST 2013
The Super Spice Dash concept is ready and goes into production.

2 SEPT 2013
The collaboration with Goodboy starts and the details of the game are worked out. Wireframing finishes.

9 SEPT 2013
The look and feel of the campaign is approved and design of the game kicks off. Razorfish works with Goodboy on the 'low poly' aesthetics.

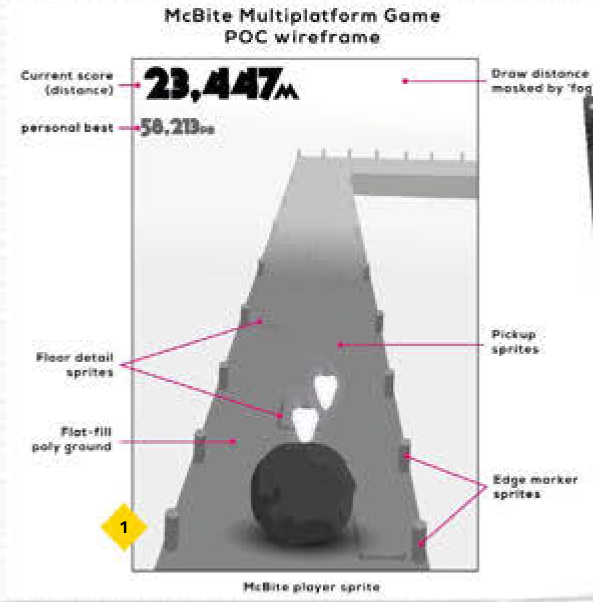
16 SEPT 2013
All POC groundwork and the initial cross-device testing completes. Design starts for level aesthetics

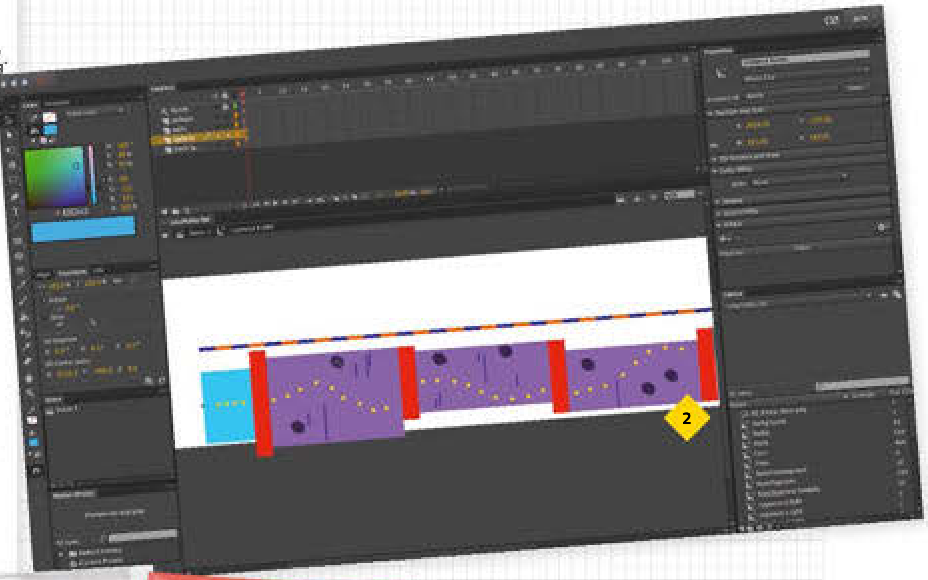
23 SEPT 2013
Development continues, adding game functionality, scoring system, timing.

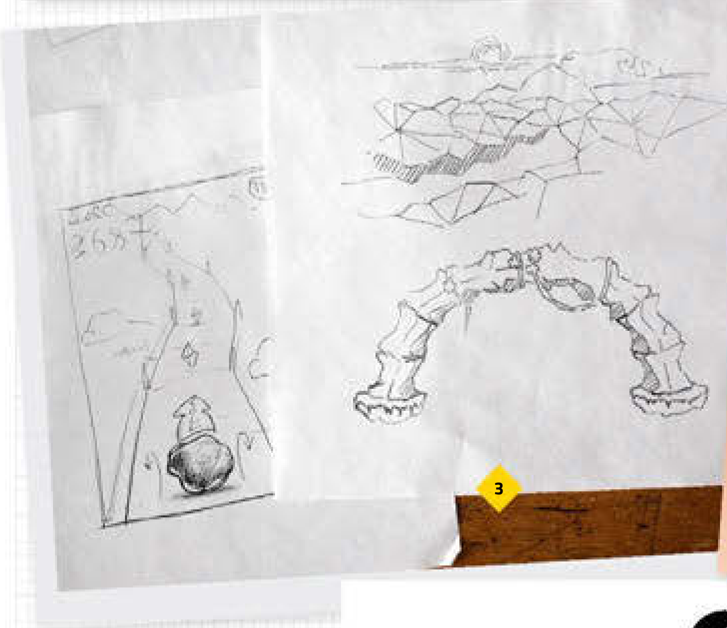
30 SEPT 2013
Development integrates level layouts and artwork as well as social scoring functionality, with final tweaks for retina mobile devices and checks on the orientation-flexible design performance.

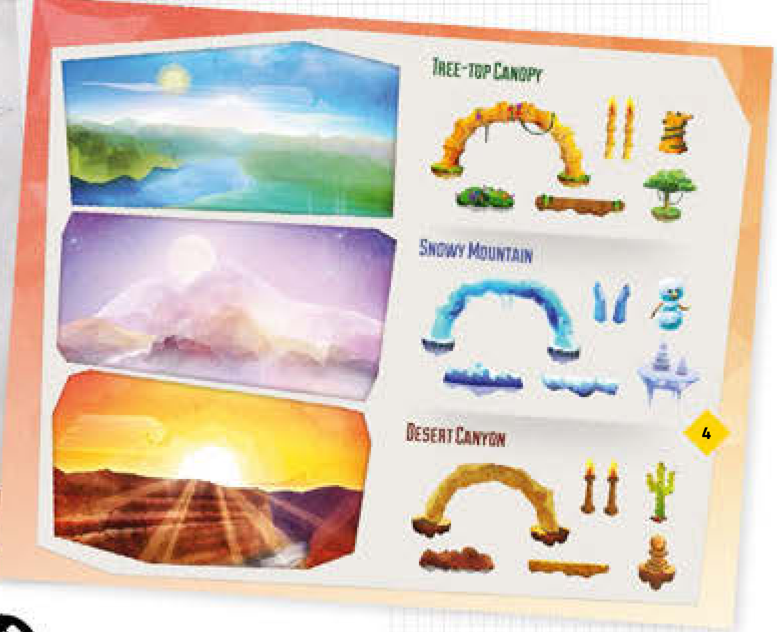
6 OCTOBER 2013
Game QA begins.

16 OCTOBER 2013
The campaign goes live and the site works on all the different devices.









EVOLUTION

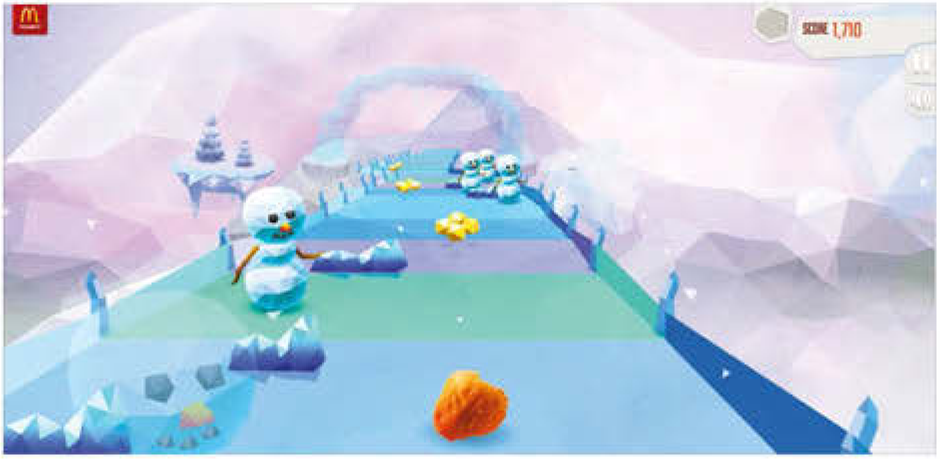
Goodboy and Razorfish discuss the evolution of the project

(1) We started by creating a wireframe to scope out the basic form of the game and the rough scope of the UI. The 3D track was definitely a big coup for a browser mobile game. The 'sketch' was mainly in the form of the original engine proof of concept. **(2)** We used Flash as a quick means of producing, testing and tweaking the level set pieces. **(3)** We have hand-drawn a series of worlds, from jungle to snowy peak mountains. We then selected the top three to be illustrated. **(4)** The three worlds that we selected are really different from one another and are all made of sprites having the same function.

Why did you opt for a gaming angle as opposed to another experience?
CL: The product is a very playful one by nature, and this spicy edition even more. We wanted to illustrate the adventures that the McBites went through to become so hot and spicy, in a fun, viral and sharable way. We wanted to combine an epic and immersive gaming experience with advertising in a playful way.

In pure technical terms, which tools, technologies and platforms did you use to build the site?
GB: Lots! I mean, there really was a lot of tech in SSD. There were regular tools such as perennial coder favourite Sublime Text 2 for getting the code down. The development team took place on a Bitbucket repo managed with SourceTree. The game's 3D was so specialised for device performance that we developed our own super lightweight engine. HUD and UI was handled by our own open source engine, Pixi.js. Flash got a look in too as, once again, we used it (well its IDE) as a convenient level editor, publishing spitting track data out direct to our server for instant cross device testing.

There was even a bit of tech applied to the visuals we created. We used a great iPad app called Tres (*netm.ag/tres-253*) by The So3, which allowed us to create the low-poly-style backdrops and in-game assets. It's actually a really cool app, and outputs either high-res PNG or SVG assets.



Icy cold Snowy Peak Mountain is the second level of the game. It's an icy world full of scary traps

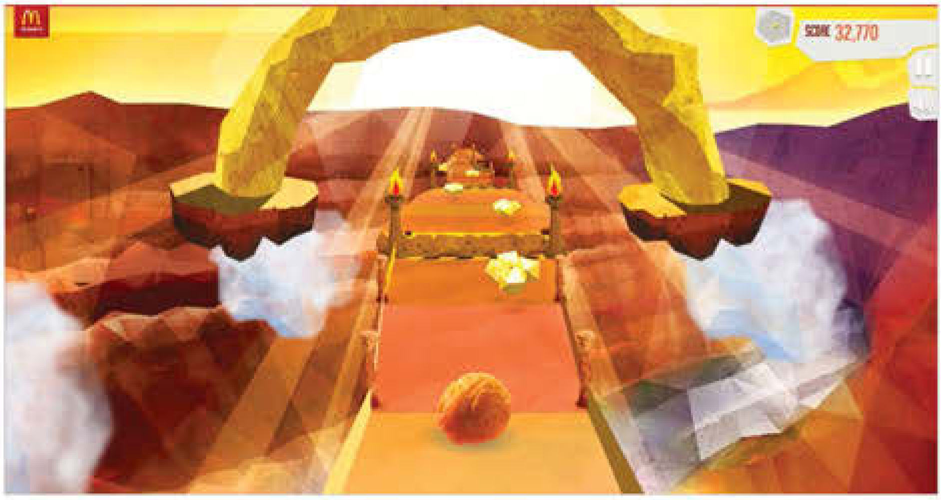
What were the biggest technical challenges that needed to be overcome?
GB: Getting the game to run on a range of mobile devices, including Android and iOS phones and tablets. Oh, and Internet Explorer always likes to throw in a few curveballs. Making a 'one-size-fits-all' solution that suits lots of patently different sizes is no easy task. Responsive design is one thing, but SSD had to dynamically cope with pretty much everything! Responsive, layout, gameplay, performance and control method. The whole nine yards.

iOS also had its own specific challenges. Namely the curse of the WebView. It will probably come as a surprise to more than just us quite how severe this issue is. Basically, any WebView that isn't iOS Safari is subject to a 3.3 times performance hit. There is no trick around it or anything like that. Only Safari gets the Nitro JavaScript engine, and when it comes to a high performance mobile browser game, that's a very big deal!

Looking over the horizon, which web technologies are you looking forward to experimenting with?
GB: As a borderline institutionalised Apple fanboy I've got to admit that the browser performance of devices like the HTC One really has been an eye opener. Not so much on this game, but seeing it throw around WebGL content with ease is quite a thing to behold! WebGL and in particular its application to 2D content is something we think has really amazing scope.

What's the feedback from the client and its customers been like?
CL: The game was very well received by the clients, and the public. The customers spent an average of more than seven minutes playing with many coming back for more fun later.

What's the secret of a good burger?
CL: Two beef patties and three-part bun.
GB: Three slices of cheese.



Burning hot Here is the third level of the game, 'Sand Storm Mirage', which is a burning hot world

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/pixi-253*

ABOUT THE AUTHOR

MAT GROVES

w: *doormat23.com*

t: @Doormat23

areas of expertise: JavaScript, visual coding, games programming

q: what's the worst you've ever screwed up at work?
a: I once made a project that was only 99 per cent awesome

PIXI.JS

GET STARTED WITH PIXI.JS

Mat Groves explains how to use 2D renderer Pixi.js to seamlessly deliver interactive content across different devices and browsers

RESOURCE

WATCH A DEMO

See a video of what we'll be making in this demo here: *goodboydigital.com/pixijs/koi_fish_pond*

Pixi.js (*pixijs.com*) is a 2D renderer. In a world gone mad on 3D we're going all flat, but not because we're anti-3D; we love those little triangles! The aim of this project is to provide a fast, lightweight 2D library that works across all devices, both mobile and desktop. The Pixi.js renderer allows everyone to enjoy the power of hardware acceleration without prior knowledge of WebGL.

WHY USE IT?

The main reason for using Pixi.js is therefore as a means of delivering awesome, interactive content that can reach as many devices and browsers as physically possible. Right now, the browser landscape is a world of fragmentation so Pixi.js is intended as a handy tool to have in your back pocket to help reach them all, and make them all play nice!

To get started, download the source files by visiting *netm.ag/pixi-253*. Open up the `index.html` file and you'll see that the basic HTML is set up for you, including the all important importing of Pixi.js:

```
<!DOCTYPE HTML>
<html>
<head>
    <title>My Pixi.js Koi Fish Pond</title>
    <style>
        body {
            margin: 0;
            padding: 0;
            background-color: #000000;
        }
        .rendererView {
            position: absolute;
            display: block;
            width: 100%;
            height: 100%;
        }
    </style>
    <script src="js/pixi.js"></script>
</head>
<body>
    <script>
        // Code goes here
    </script>
</body>
</html>
```

First, we'll need to create an instance of a Pixi.js renderer. Pixi.js has two renderer flavours: WebGL and Canvas. The best way to create a renderer is to use the `autoDetectRenderer` function. Pixi.js will perform an internal check and make sure to return a renderer most suitable to the user's browser:

```
var viewWidth = 630;
var viewHeight = 410;
var renderer = PIXI.autoDetectRenderer(viewWidth,
viewHeight);
renderer.view.className = "rendererView";
```

Now we have a renderer, we add its view to the DOM:

```
document.body.appendChild(renderer.view);
```

It's time to create a Pixi `Stage` element. Much like Flash, the `Stage` element is the root display object where all visual elements will be added. The only parameter for creating a stage is its background colour. Let's use white!

```
var stage = new PIXI.Stage(0xFFFFFF);
```

To add some visuals to our new stage, we need a `Texture` and a `Sprite`. A `Texture` stores the information that represents an image. It cannot be added to the display list directly and so has to be mapped onto a `Sprite`. `Texture.fromImage` tells Pixi.js to create a new `Texture` based on the image path provided, which can be reused for multiple `Sprites`. Now we create a `Sprite` that will use it:

```
var pondFloorTexture = PIXI.Texture.fromImage("img/
pondFloor.jpg");
var pondFloorSprite = new PIXI.Sprite(pondFloorTexture);
```

Once a `Sprite` has been created, for it to be rendered, it must be added to the `Stage` display list:

```
stage.addChild(pondFloorSprite);
```

Done! If you were to render the stage now you would see a nice image of a rockpool floor on your screen. If

IN-DEPTH

HAVE YOUR CAKE AND EAT IT

Reaching tablets and mobiles while also tapping into the power of WebGL can seem something of a dichotomy. On the one hand, you have relatively modest GPUs (although getting better every day) of mobile devices and on the other you have technology designed expressly to tap into the power afforded by the GPU grunt of modern 'desktop' computers.

I started work on Pixi.js expressly for that purpose. To allow the creation of content that can utilise WebGL's amazing performance, but to do so in a fashion that means it can fall back to Canvas in a seamless manner, delivering the same core content to mobile devices and non-WebGL browsers.

While the Canvas fallback inevitably lacks some of the graphical features of the WebGL iteration, the thing Pixi offers is a single code base that can harness the abilities of WebGL when they're available, but deliver the same content via Canvas without additional effort or knowledge.

In much the same way as we now accept adaptive design as a staple feature of UI aesthetics, I wanted Pixi.js to do the same thing for visual effects and fidelity. Where it's available, WebGL can push the boundaries and enable insane new tricks to play with. However, the Canvas fallback is always on hand to filter the experience back to devices playing catch up in the GPU arms race.

Eat cake Full-on eye candy in WebGL with full functionality Canvas fallback

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

FOCUS ON

PIXI IN PRACTICE

The thing I get asked most about with Pixi.js is when we're going to add particular physics features or collision systems. Pixi.js is a renderer rather than a games engine. Much like Flash, Pixi.js is simply there to create interactive content. How and what people do with it is up to them. As an example, we made Run Pixie Run (*netm.ag/runpixie-253*) as our first demo of what Pixi can do. In this instance we used it on render duties to make a slick, all-device browser game. We recently worked with Razorfish to make the Saver Menu (*netm.ag/saver-253*) site as part of McDonald's latest campaign, which is all about a tight, interactive experience site, tailored for desktop and tablet.

WHAT NEXT?
Done with the fishies? Here's a few resources to get you going:

Pixi.js homepage
For inspiration and examples, visit the home of Pixi.js (*pixijs.com*)
The nuts, bolts ... and forks!
See the GitHub repository: *github.com/GoodBoyDigital/pixi.js*
Goodboy archives
Where I go to talk shop: *www.goodboydigital.com/category/pixi-js*
HTML5 Game Devs forum
Where everyone talks: *www.html5gamedevs.com/forum/15-pixijs*

Run Pixie Whether a game like Run Pixie Run or an interactive campaign, Pixi.js gives you the power to deliver to all devices

you come from the Flash side of the world then you will know where Pixi.js gets a lot of its terminology. Now we have a rockpool, we need to create 20 swimming fish and add them to the stage:

```
// set how many fish we would like to add
var totalFish = 20;
// create an array to store a reference to the fish in the
pond
var fishArray = [];
for (var i = 0; i < totalFish; i++)
{
// generate a fish id betwen 0 and 3 using the modulo
operator
    var fishId = i % 4;
    fishId += 1;
    // generate an image name based on the fish id
    var imagePath = "img/fish"+fishId+".
png";
    // create a new Texture that uses the image name that
we just generated as its source
    var fishTexture = PIXI.Texture.fromImage(imagePath);
    // create a sprite that uses our new sprite texture
    var fish =  new PIXI.Sprite(fishTexture);
    // set the anchor point so the fish texture is centred on
the sprite
    fish.anchor.x = fish.anchor.y = 0.5;
    // set a random scale for the fish - no point them all
being the same size!
    fish.scale.x = fish.scale.y = 0.8 + Math.random() * 0.3;
// finally let's set the fish to be a random position..
    fish.position.x = Math.random() * viewWidth;
fish.position.y = Math.random() * viewHeight;
    // time to add the fish to the pond container!
    stage.addChild(fish);
```

We created more sprites and modified the sprite properties. Every Pixi.js sprite has a position, scale and rotation property which you can modify to move them around. We now need to add a few extra properties to make the little guys swim around:

```
    // create a random direction in radians. This is a number
between 0 and PI*2 which is the equivalent of 0 - 360
degrees
    fish.direction = Math.random() * Math.PI * 2;
    // this number will be used to modify the direction of the
fish over time
    fish.turningSpeed = Math.random() - 0.8;
    // create a random speed for the fish between 0 - 2
    fish.speed = 2 + Math.random() * 2;
    // finally we push the fish into the fishArray so it it can
be easily accessed later
    fishArray.push(fish);
}
```

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL

Let's create a bounding box for the fish. We'll use this to ensure, when the fish swim out of the bounds, they wrap around the scene. The padding ensures the fish are off screen before it wraps to avoid any 'popping'.

```
// create a bounding box for the little fish
var fishBoundsPadding = 100;
var fishBounds = new PIXI.Rectangle(-fishBoundsPadding,
            -fishBoundsPadding,
            viewWidth + fishBoundsPadding * 2,
            viewHeight + fishBoundsPadding * 2);
```

The waves are created using another Pixi.js object called a TilingSprite. Using this, we can scroll the waves over the pond to give a nice water effect. We create a new texture and create a new tiling sprite:

```
var waveTexture = PIXI.Texture.fromImage("img/waves.
png");
var wavesTilingSprite = new PIXI.TilingSprite(waveTexture,
viewWidth, viewHeight);
wavesTilingSprite.alpha = 0.2;
stage.addChild(wavesTilingSprite);
```

We then set the alpha of the object. This is a Pixi.js property that sets how opaque a display object is. Now add our new waveSprite to the stage. Only one more thing to set up now! The displacement filter will create the real time wobbly water effect. (Note: this is a WebGL-only feature.)

```
// create a new wave texture to add over the fish
var waveDisplacementTexture = PIXI.Texture.
fromImage("img/displacementMap.jpg");
var displacementFilter = new PIXI.DisplacementFilter(wave
DisplacementTexture);
// configure the displacement filter..
displacementFilter.scale.x = 50;
displacementFilter.scale.y =
50;
// apply the filters to the stage
stage.filters = [displacementFilter];
```

It's time to create the update loop to run each frame and update the position of the fish, water and displacement. Use the requestAnimationFrame, like so:

```
var tick = 0;
requestAnimationFrame(animate);
function animate()
{
```

We loop through the fish array and update the position of the fish based on speed and direction. The fish are wrapped as they reach the edge of the screen.

```
    // iterate through the fish and update the position
    for (var i = 0; i < fishArray.length; i++)
    {
        var fish = fishArray[i];
        fish.direction += fish.turningSpeed * 0.01;
        fish.position.x += Math.sin(fish.direction) * fish.
speed;
        fish.position.y += Math.cos(fish.direction) * fish.
speed;
        fish.rotation = -fish.direction - Math.PI/2;
        // wrap the fish by testing there bounds..
        if(fish.position.x < fishBounds.x)fish.position.x +=
fishBounds.width;
        else if(fish.position.x > fishBounds.x + fishBounds.
width)fish.position.x -= fishBounds.width
        if(fish.position.y < fishBounds.y)fish.position.y +=
fishBounds.height;
        else if(fish.position.y > fishBounds.y + fishBounds.
height)fish.position.y -= fishBounds.height
    }
```

Let's update the tiling water texture. A TilingTexture has an extra property called tilePosition that controls the offset of the texture as it tiles. Increasing this on each frame will give the illusion of the water moving:

```
    // increment the ticker
    tick += 0.1;
    // scroll the wave sprite
    wavesTilingSprite.tilePosition.x = wavesTilingSprite.
tilePosition.y = tick * -10
```

We also need to update the displacement map offset. Moving this each frame will scroll the displacement map giving the lovely illusion of rippling water.

```
    // update the displacement filter by moving the offset of
the filter
    displacementFilter.offset.x = displacementFilter.offset.y
= tick * 10
```

We need to render the scene using our renderer. Calling this function will draw all of the contents of the stage to the renderer's view that we attached to the stage. Once the scene has been rendered we requestAnimationFrame again. This creates a loop of constant updating and rendering:

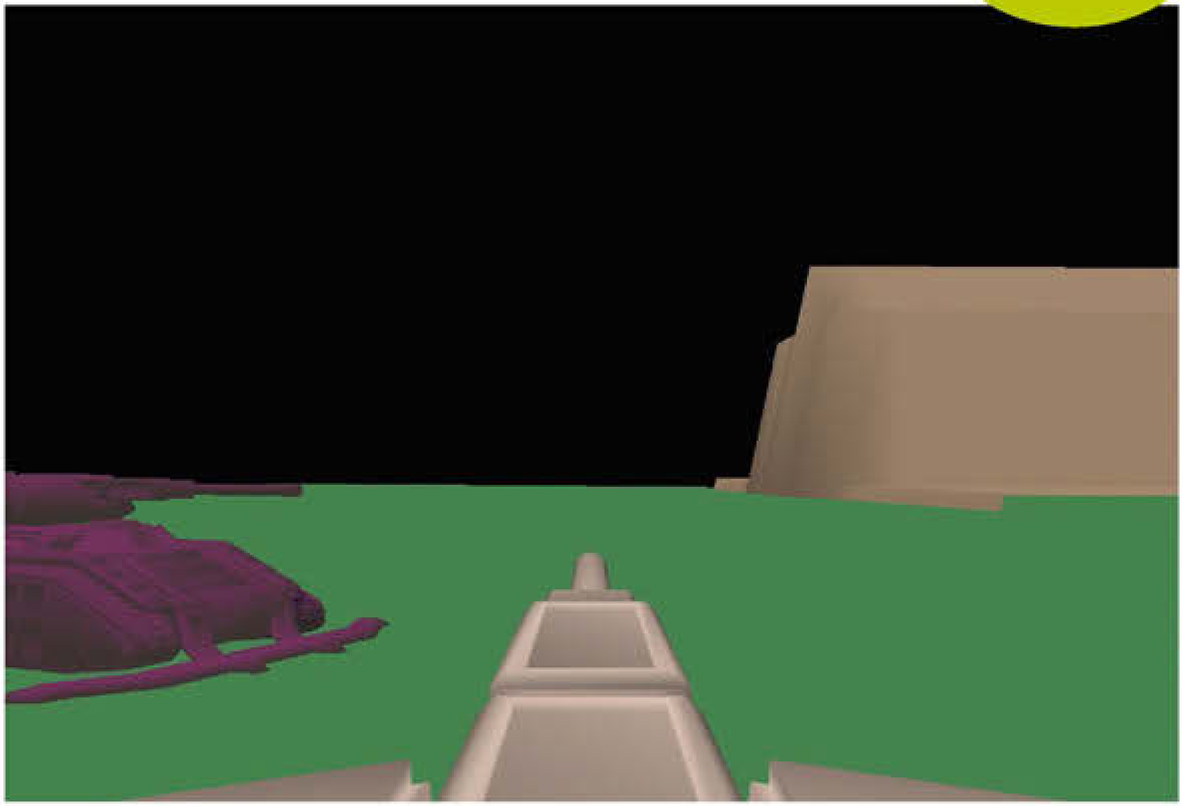
```
    // time to render the state!
renderer.render(stage);
// request another animation frame
requestAnimationFrame( animate );
}
```

Done! Don't be 'koi', load it up and try it out.

Download the files here!
You'll find all the files you need for this tutorial at *netm.ag/game-264*

ABOUT THE AUTHOR
JAMES WILLIAMS

w: *jameswilliams.be/blog*
t: @ecspike
job: Course developer, Udacity
areas of expertise: HTML5, Java, JavaScript, Groovy, app and game develpment
q: what's the first gadget you owned?
a: Sega Game Gear



THREE.JS

BUILD A BASIC COMBAT GAME WITH THREE.JS

James Williams explains how to use 3D graphics library three.js to build a tank combat game that runs in a browser via WebGL

three.js (*threejs.org*) is a 3D graphics JavaScript library that helps simplify the process of creating scenes with WebGL. Together, three.js and WebGL have been used on projects ranging from online advertising campaigns for *The Hobbit* movie trilogy to visualisations for Google I/O. In this article, we'll use three.js to create a simple game. There isn't space to provide a complete step-by-step guide, but I will introduce the key concepts. Once you've mastered them, the complete source code is provided at *netm.ag/game-264* for you to explore in detail.

Back when I was teaching programming classes at a computer camp, a popular multiplayer game amongst the kids was Recoil. In it, you control an armoured tank and, most importantly, blow up stuff. It will be the inspiration for our game.

THE BASIC SETUP

Below is the code to set up a basic scene containing a camera and a light:

```
var height = 480, width = 640, fov = 45, aspect, near, far;
aspect = width/height;
near = 0.1; far = 10000;

self.renderer = new THREE.WebGLRenderer();
self.renderer.setSize(width, height);
```

VIDEO

James Williams has put together an exclusive screencast to go with this tutorial. Watch along at *netm.ag/threejsvid-264*

Our inspiration Recoil was a tank game developed in 1999 by Zipper Interactive. We'll use it as the basis for our own game

```
self.camera = new THREE.PerspectiveCamera(fov, aspect,
near, far);
self.camera.position.y = 5;
self.camera.position.z = 30;

var light = new THREE.DirectionalLight(0xFFFFFF, 0.75);
light.position.set(0,200, 40);

self.scene = new THREE.Scene();
self.scene.add(this.camera);
self.scene.add(light);

document.querySelector('#c').appendChild(this.renderer.
domElement)
```

LOADING MODELS

Although creating objects using code is fun, for anything complex you are going to want to use specialist 3D modelling software. For this game, I'm using Blender, a very capable and mature 3D application (see boxout opposite). three.js supports a couple of common 3D file formats natively, and there are plugins for applications like 3ds Max, Maya and Blender that will enable you to export models in a JSON format that three.js can parse more easily.

There are lots of models available for free on websites like Blend Swap (*blendswap.com*) and Blender Artists (*blenderartists.org*). three.js supports import of both static and dynamic models. The latter require a bit more work, both before and after you import them into your game. Rigging – preparing a model for animation – is outside the scope of this article. However, the models on sites like Blend Swap often come pre-rigged so you can create your own animations; or even with their own set of animations already created.

▶

FOCUS ON

WHAT IS BLENDER?

Blender (*blender.org*) is a free and open-source 3D modelling and animation application. Blender boasts the distinction of being one of the earliest crowdfunded software applications, if not the first. When the controlling company went bankrupt in 2002, the community raised €100,000 to purchase the copyrights and free the source code.

It has a feature set that is comparable to commercial software such as Autodesk's 3ds Max and Maya, and has been used to create animations and visual effects for films and television (*Spider-man 2*, History Channel programmes and others), and games – it actually includes its own game engine. NASA even publishes the 3D models of its various spacecraft and satellites in Blender format.

Blender's core written is C and C++ but it supports scripting in Python to provide new functionality. New file formats, tools, game logic and automated tasks are added through the Python APIs. Support for *three.js import/export* is provided via a Python plugin (included in the three.js GitHub repo).

The Blender Foundation oversees the development of Blender. In addition to hosting a yearly conference featuring an animation festival of short films created using Blender, it also sponsors Open Projects. The goal of these projects is to validate the use of Blender as a media creation tool, expose gaps in features and close them. These films or games are community-developed using Blender and all assets are licensed under Creative Commons. The features used will eventually make into a new version of Blender.

Bunny beginnings *Big Buck Bunny*, originally known as 'Project Peach', is an open source animated comedy film sponsored by the Blender Foundation

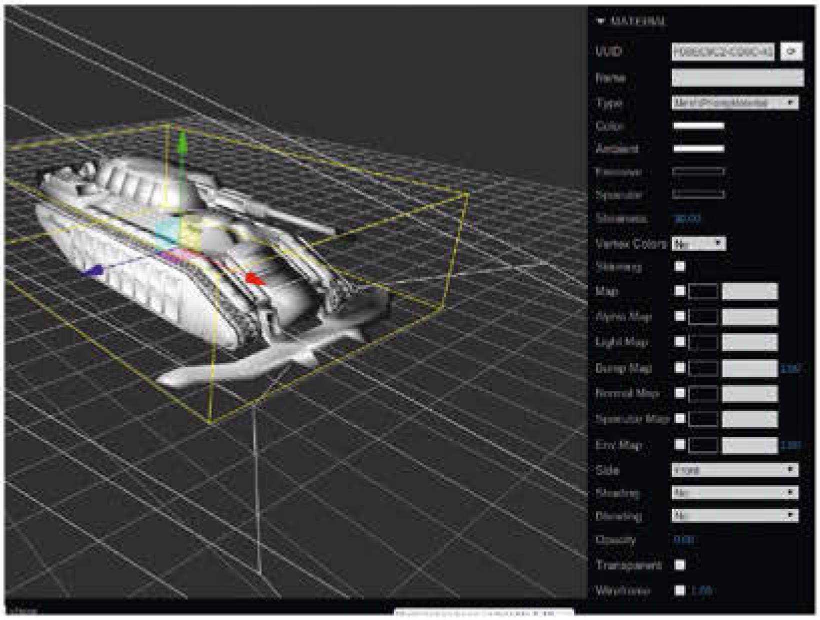
▶ three.js uses a JSONLoader object to import models. This object includes a load function that takes a URI fragment pointing to a JSON file, a callback function, and an optional fragment pointing to assets:

```
var scope = this;
var loader = new THREE.JSONLoader();
loader.load('model/chaingunner.json', function(geometry, 
material) {
   var texture = THREE.ImageUtils.loadTexture("model/
chaingunner_body.png");
   var material = new THREE.MeshLambertMaterial({color:0xF
FFFFF, map:texture, morphTargets: true});
   scope.human2 = new THREE.MorphAnimMesh(geometry, 
material);
   scope.human2.position.y = 0.4;
   scope.human2.position.x = 8;
   scope.human2.scale.set(0.5, 0.5, 0.5);
   scope.scene.add(scope.human2);
 });
```

TEXTURES AND MATERIALS

Before we talk about the mesh we've loaded, let's talk about how it will be textured. three.js has a helper object THREE.ImageUtils to load textures in a single line of code. Once the texture is loaded, you can set properties on it to determine how it will be mapped to an object, before adding it to a material. A material determines the appearance of an object. Depending on the material, it also determines how light sources affect and interact with the object.

MeshBasicMaterial doesn't consider any of the lights that may be present. MeshLambertMaterial and MeshPhongMaterial do take lighting into account. Lambert-shaded surfaces are generally diffuse. Phong shaded surfaces are calculated on the pixel level and enable the user to set components to determine how shiny a material is. Phong materials can do anything that Basic or Lambert materials can do, with more granular control.



Experimenting Tank model shown in the three.js editor with materials and lighting. The editor is a great place to experiment with materials

ShaderMaterial, which I'll leave you to explore on your own, gives you the greatest control of all. It uses GLSL, a C-like language that runs on the GPU and enables you to interact with every vertex and pixel in a material.

MORPH TARGETS AND ANIMATION

The model we are using comes pre-rigged, and has some animations attached. three.js allows you to animate objects using either morph targets or skeletal animation.

For skeletal animation, an artist sets up a series of 'bones' in a 3D application like Blender. These form a digital equivalent of the armature inside a stop-motion model. When the bones are animated, they deform the 3D mesh surrounding them, meaning that the model itself moves. This bone data can be exported to three.js format.

three.js supports both static and dynamic models. You can find free 3D models online

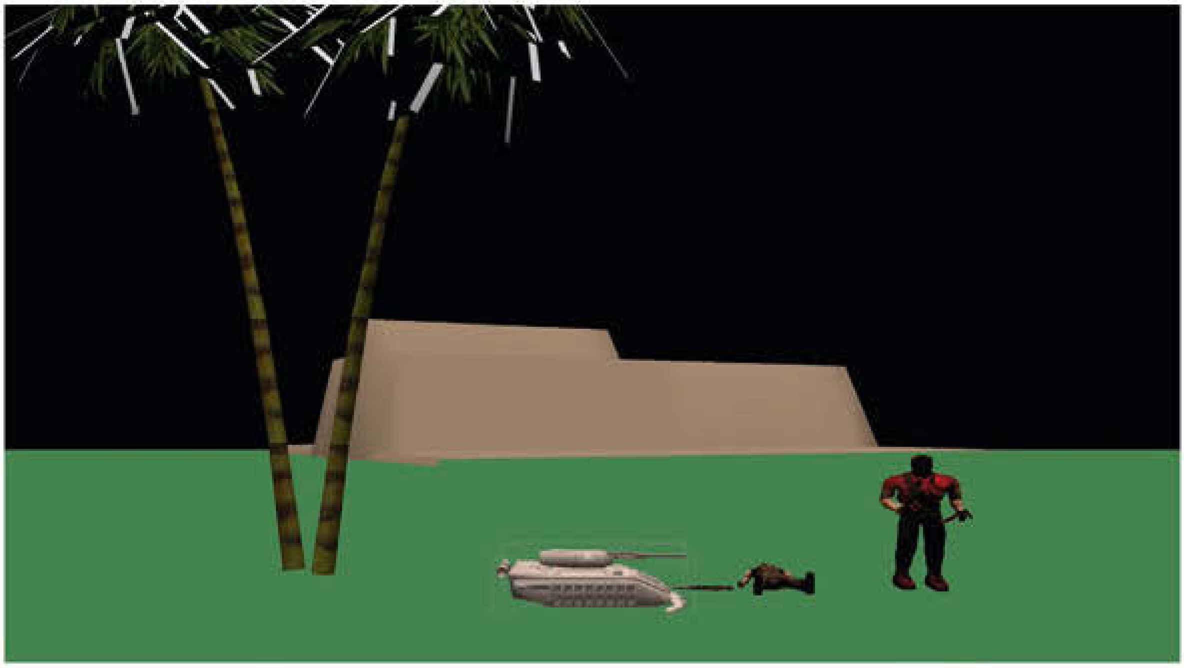
Morph targets, on the other hand, store the positions of every vertex in the mesh directly, for every keyframe in your animation. So if you have a model with 500 vertices – by no means unreasonably large – and you have 10 morph targets, each corresponding to its shape at a keyframe in the animation, you are storing data for 5,000 extra vertices. Morph targets are more reliable to work with, but they balloon file size.

MorphAnimMesh is a special type of THREE.Mesh that loads all the vertex data for the mesh and enables you to run animations selectively. Alternatively, if a mesh only has a single animation, you can create a MorphAnimation object.

```
   human2.animation = new THREE.MorphAnimation( scope.
human2 );
   human2.animation.play();
```

HANDLING KEYBOARD INPUT

THREEx (*threejsgames.com/extensions*) is a set of third-party extensions to three.js. It includes threex.



Making motion An animated model in the middle of its animation cycle. Another model is resting in its 'dead' pose

VIDEO

Watch James Williams' talk Introduction to WebGL and three.js at: *netm.ag/WebGLintro-264*

keyboardstate, an extension that keeps track of keys pressed on the keyboard. At first glance, this doesn't seem that novel, but it supports multiple simultaneous keystrokes: for example, the use of modifier keys (shift, alt, ctrl, meta).

Traditionally, you would write code that adds event listeners on `keyup` or `keydown` and immediately execute code. In a game, you want to update only once per frame. Threex.keyboardstate uses the `keyup` and `keydown` listeners to update a map object which you are free to query as frequently as you want, to determine the state of a key.

The following code sets up the extension and uses it to determine the `pressed` state of a key:

```
var keyboard = new THREEx.KeyboardState();
// ...
if (keyboard.pressed('W')) {
  this.tank.translateX(moveDelta);
}
```

ANIMATING ON A PATH

Animating a mesh in place isn't very exciting. Let's give some of our objects autonomous movement to make the game more engaging. To do this, I created a 'class' called `PathAnimation`. It is largely derived from the `MorphAnimation` class, but instead of taking a set of arrays of vertices representing a mesh, it takes a set of points that a mesh will move along.

three.js has a couple of options to create and specify path information. I wanted a path that looked somewhat realistic, so I used `EllipseCurve` and its `getPoints` function to calculate a set of points:

```
var curve = new THREE.EllipseCurve(
  0,  0,            // ax, aY
  200, 200,         // xRadius, yRadius
  0,  2 * Math.PI,  // aStartAngle, aEndAngle
  false             // aClockwise
 );
var points = curve.getPoints(100);
var pathAnim = new PathAnimation(mesh, points, 0.4);
```

The following code checks if the animation is playing (i.e. the object is moving along the path). If so, it adds the delta to the current time, uses this to determine at which point the mesh should be located and moves it to that point. If the current time is larger than the duration of the animation, it starts again:

```
update: function(delta) {
  if (this.isPlaying === false || this.mesh == undefined) return;
  this.currentTime += delta;
  if (this.loop === true && this.currentTime > this.duration) {
   this.currentTime %= this.duration;
  }
  var interpolation = this.duration / this.points.length;
  this.point = Math.floor(this.currentTime / interpolation);
  var vectorCurrentPoint = this.points[this.point];
  if (vectorCurrentPoint)
   this.mesh.position.set(-vectorCurrentPoint.x, this.y,
vectorCurrentPoint.y);
 }
```

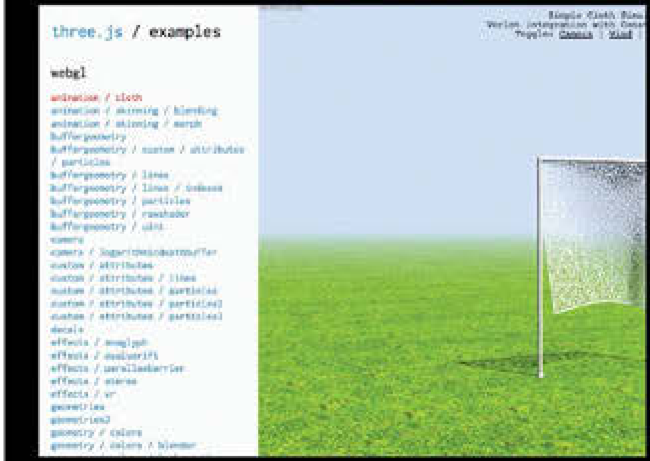
►

RESOURCES

FURTHER READING

three.js is a rapidly evolving library and it is not uncommon for things to be deprecated or change in a way that breaks your demos. As a result, while a few resources are available in dead-tree form, many of them are already obsolete (or on their way to becoming so). Your best bet for up-to-date information is to use internet resources. Things still break, and often, but unlike books, blog posts stand a reasonable chance of being updated.

- Threejs.org (*threejs.org*) – The definitive source for most things three.js. The official documentation sometimes has holes but almost all of the demos allow you to view the source code to reuse in your own apps
- THREEx (*threejsgames.com/extensions*) – In this article, I used only threex.keyboardstate, but there are many other useful three.js extensions in the THREEx collection
- threejsboilerplate (*netm.ag/boilerplate-264*) – Jerome Etienne, creator of the THREEx extensions, has a starter project that does all the mundane stuff (setting up the scene, camera and binding some simple controls) for you
- Shadertoy (*shadertoy.com*) – This is the best place to experiment with GLSL shaders. We didn't cover them in this article, but they are a key part of any three.js/WebGL project, as they control the way surfaces react to light, and therefore how they look
- Aerotwist tutorials (*aerotwist.com/tutorials*) – Paul Lewis has published some tutorials on shaders and particle systems


three.js / examples
webgl
animation / cloth
animation / skinning / blending
animation / skinning / morph
buffergeometry
buffergeometry / custom / attributes / particles
buffergeometry / lines
buffergeometry / lines / indexed
buffergeometry / particles
buffergeometry / rawshader
buffergeometry / uint
camera
camera / logarithmicdepthbuffer
custom / attributes
custom / attributes / lines
custom / attributes / particles
custom / attributes / particles2
custom / attributes / particles3
decals
effects / anaglyph
effects / oculusrift
effects / parallaxbarrier
effects / stereo
effects / vr
geometries
geometries2
geometry / colors
geometry / colors / blender
Simple Cloth Simulation
Verlet integration with Constraint relaxation
Toggle: Camera | Wind

Advanced capabilties Three.js supports more advanced physics than we use in our game. This cloth demo is from *threejs.org*

COLLISION DETECTION

Now we have our animations wired up, we need a way to inform characters they should respond to being hit. It's possible to get very precise responses with collision detection by using advanced algorithms and physics formulae, but for our simple game, we can use bounding volumes – boxes and spheres – to determine if two objects have touched. These are simple geometric forms that surround a mesh, enclosing all of its vertices.

Luckily, three.js has bounding volumes built into the library, so they are automatically generated for an object. `BoundingBoxHelper` allows you visualise the position of a bounding box, via this code:

> In order to retain a frame rate of 60fps, your game has 16.6ms to complete all of its calculations

```
var bbox = new THREE.BoundingBoxHelper( tank );
bbox.update();
```

You will have to update your bounding box on each frame of the animation to make sure it stays in sync with your model.

But why do we need bounding volumes? In order to retain a playable frame rate of 60 frames per second, your game has roughly 16.6ms to complete all the calculations it needs, compute the new positions of objects, and redraw them. Checking for collisions between two complex meshes, each of which may contain thousands of faces, is computationally very

Colliding figures Bounding volumes can be used to visually check collisions and to better understand how cameras and lights interact within the scene

Combat view An enemy tank viewed through the first-person perspective of the player

RESOURCES

Read a sample chapter from James Williams' book *Learning HTML5 Game Programming* at: *netm.ag/samplechapter-264*

costly. But by checking for collisions between their bounding volumes, we can obtain a reasonable approximation of whether the meshes themselves have collided – and checking for box-box or box-sphere intersections is relatively fast.

An object can have several bounding volumes attached to it. For example, for a human-like character, you could have individual bounding volumes around the head, torso, arms, legs, hands and feet. There might also be a bounding sphere around the entire character.

As the game runs, it checks first to see if it should check for collisions between two objects. Let's say that we only want collisions to be computed if the objects are closer than 50 units. If we know the centre of mass for both objects, we can quickly determine if we should move further.

If that check passes, we go on to check the bounding spheres to see if a collision has occurred. If this is the case, we can select a subset of bounding volumes to determine – in the case of our character – which particular part of the body has been struck. This way, we only run costly computations when we really need to.

TRACING RAYS

The `ray` class is similar to firing an arrow at a bunch of objects and asking which ones were hit:

```
this.ray = new THREE.Raycaster();
```

```
checkCollisions: function(b, vec, objects) {
  this.ray.set(b.position.clone(), vec);
  var collResults = this.ray.intersectObjects(objects, true);
  if (collResults.length > 0 && collResults[0].distance < 5) {
    var object = collResults[0].object;
    // Do something with that collision
    this.removeBullet(b);
    return object;
  }
}
```

If the `ray` is used to represent a photon of light rather than a physical object, this can be used to simulate the way light rays bounce off or interact with surfaces they strike, enabling us to set up photorealistic illumination in our scene – a concept known as ray tracing. Ray tracing is computationally very costly, which is the reason that in the code above, we worked with only a single ray and specified its direction.

GOING FURTHER

We've covered a lot of ground in this article, introducing many of the basic concepts of game development using three.js. It's a good start, but we're still a long way from a complete game. If you want to go further, you can find the complete code at *netm.ag/game-264*, and repurpose, remix or refactor whatever you need into your own games. The repo also includes a list of links to the 3D models I used. Have fun!

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/matterjs-255*

ABOUT THE AUTHOR

LIAM BRUMMITT

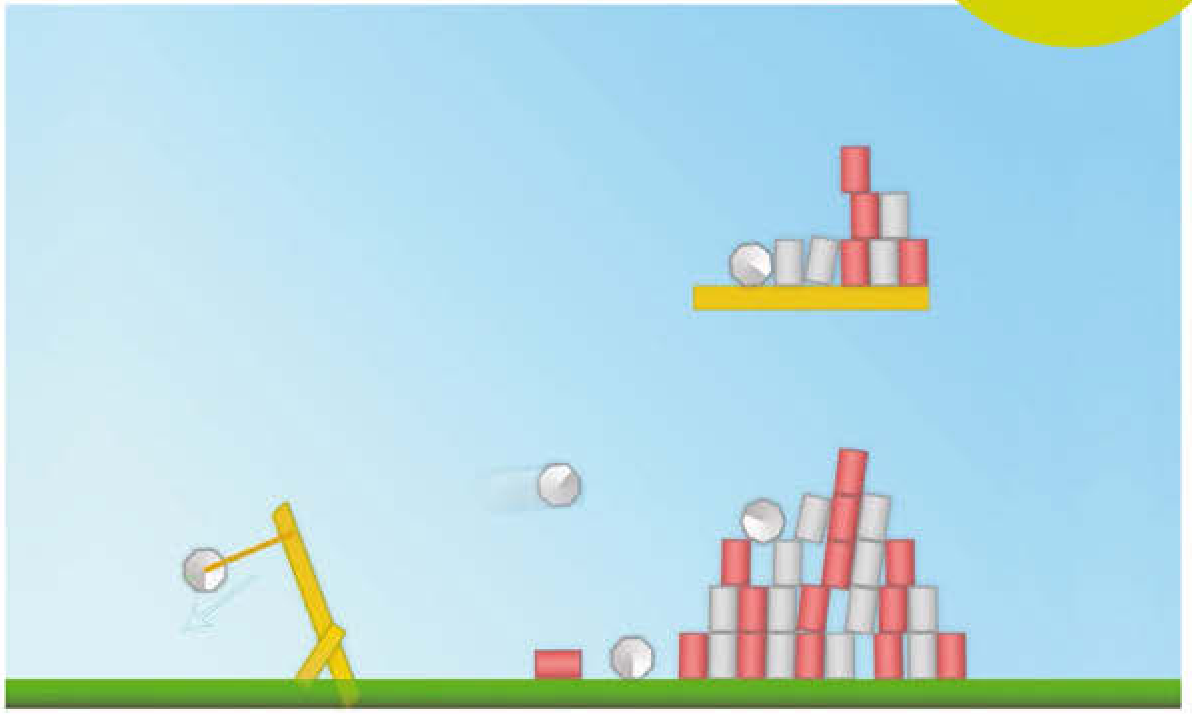
w: *brm.io*

t: @liabru

areas of expertise: HTML5, CSS3, JavaScript, app and game development

q: When was the last time you cried?

a: The last time I heard "IE6 support required"!



HTML5

BUILD AN HTML5 GAME WITH MATTER.JS

Liam Brummitt explains how to use his JavaScript physics engine, Matter.js, to create a simple web game built on HTML5 technology

Matter.js is a flexible rigid-body 2D engine with a focus on high performance, stability, ease of use and cross-platform compatibility. Although it isn't intended to be a complete game engine – more a component of one – it includes most of the things you need to get started, including a game loop, mouse controller, collision system and a renderer.

The project is currently in alpha and comes with all the usual caveats – like not using it on critical productions – but it is still a good fit for indie-style 2D games that don't require a heavyweight engine.

"But I'm already using the WhizzBang2000 engine," I hear you say! No problem. Since Matter.js includes an event callback system and configurable controllers, it should be possible to integrate the physics engine into whatever JavaScript game engine or rendering library you wish to use.

The engine includes an example canvas-based renderer, with basic support for both primitive (vector) rendering and sprite-based (texture) rendering of the physical bodies in your game world. If you need to accomplish anything further, it's easy to use this as the basis for your own renderer.

Or if canvas is a bit too pedestrian for your tastes, and you'd prefer the bleeding-edge power of WebGL, you're in luck: swap in the included example WebGL renderer (which makes use of the superb Pixi.js library – we covered it in issue 253) and you're good to go. It's possible to extend this scene-graph-based renderer to use something like three.js too.

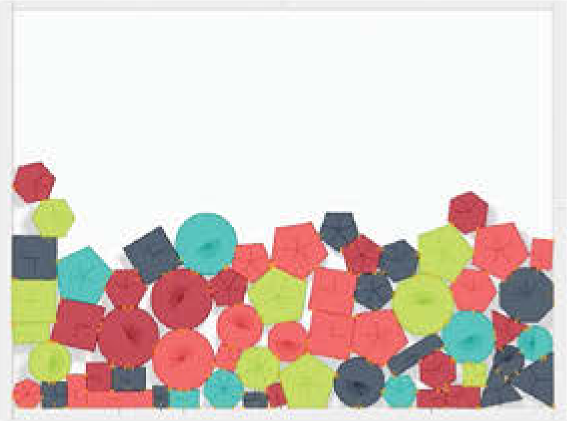
BUILDING A SIMPLE GAME

It's about time I showed you how to get up and running with Matter.js. We'll set up a simple scene:

RESOURCE

MORE INFO

You can find out more about Matter.js on the engine's project page at *brm.io/matter-js*



Scalable engine Matter.js can handle plenty of simulated rigid bodies

a pyramid of blocks, a slingshot and a boulder we'll be able to lob and smash it all down with.

If at any point you get stuck, or something doesn't make sense, you can always check the engine examples and API docs over at the project page (*brm.io/matter-js*).

First things first: grab a copy of the engine from the project page, and create a blank HTML page set up with a `script` reference to Matter.js. Open another `script` block ready to enter the code below.

You can integrate Matter.js into any other JavaScript game engine or rendering library

When you include Matter.js, the `Matter` object becomes globally available, which is where all the Matter functions live. To keep things short, I like to start by aliasing the modules I need to use at the top of the script, like so:

```
var Engine = Matter.Engine,
    World = Matter.World,
    Bodies = Matter.Bodies,
    Constraint = Matter.Constraint,
    Composites = Matter.Composites,
    MouseConstraint = Matter.MouseConstraint,
    Events = Matter.Events;
```

Create an empty function called `init`. You will put most of the code here. It is very important to make sure that after the `init` function is declared, you bind it to the window load event – otherwise you won't see anything!

```
window.addEventListener('load', init);
```

▶

★ FOCUS ON

WHY HTML5 PHYSICS GAMES?

As we've seen from the success of physics-based games on mobile app stores, physics is often a major component of modern games, with many using it as their core mechanic. Why? There just seems to be something so satisfying about building and destroying things!

A few years ago, the idea of creating real games on web-based technology would have seemed crazy. Fortunately, web technology is rapidly becoming a viable way to get your content onto a number of platforms, without breaking (too much of) a sweat.

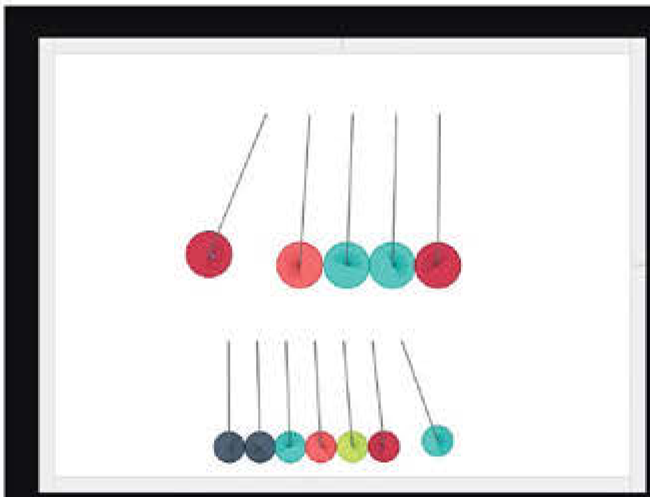
In recent years there have been massive advances in HTML5 technologies, from the performance of core JavaScript engines (V8, SpiderMonkey, and so on) to APIs such as canvas and WebGL.

And it's only going to keep getting better. Recently, Mozilla has been working with both Unity (*netm.ag/unitywebgl-255*) and Unreal Engine (*netm.ag/unrealwebgl-255*) to get them working at near-native speed in the browser, with the help of asm.js: a highly optimised version of JavaScript. Now that's impressive.

Matter.js may not be quite in this league – it isn't a fully featured game engine like Unity or Unreal – but it may still be a good fit for indie-style 2D games that don't require a heavyweight engine.

Online play The popular Unity game engine is adding WebGL export in version 5

Tools & technologies
Gallery
The future of JavaScript
Frameworks
Performance & workflow
UI & RWD
WebGL



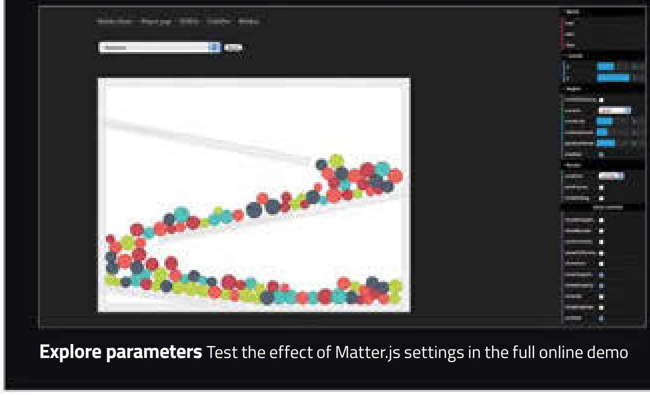
RESOURCES

DEMO SCENES

You can currently see 20 different examples and demos on the Matter.js project page (*brm.io/matter-js*), including the classic Newton's cradle above, which – as well as being slightly hypnotising – demonstrates the momentum-conservation properties of the engine.

There's also a mobile-specific demo, which includes code for shifting gravity based on the device's tilt sensors. While this currently only works smoothly on higher-end devices, it shows the potential for mobile use, even at this early stage.

The demos on the project page use simple white-on-black outline graphics, but you can see more colourful versions on CodePen (*codepen.io/liabru*) or visit the full online demo (*brm.io/matter-js-demo*) and play with the renderer settings for yourself.



Explore parameters Test the effect of Matter.js settings in the full online demo

First, let's create an engine, passing the element where the canvas is to be inserted:

```
var engine = Engine.create(document.body);
```

We want to control the scene with the mouse. The engine includes a helpful constraint called `MouseConstraint`, so we'll add it now:

```
        var mouse = MouseConstraint.create(engine, {
    constraint: { stiffness: 1 }
});
```

Here I've passed the `engine`, which is required for the `MouseConstraint` to work. I've also passed an options object, which changes the constraint's `stiffness` to make it a little less slack than the default setting.

Let's add some bodies to the scene. First of all, we need a ground object, otherwise objects will fall out of the world!

```
var ground = Bodies.rectangle(395, 600, 815, 50, { isStatic:
true });
```

Here the `Bodies` module is a factory that can create common types of geometric object: squares, rectangles, circles, triangles and other polygons. I've passed the position and dimensions we require.

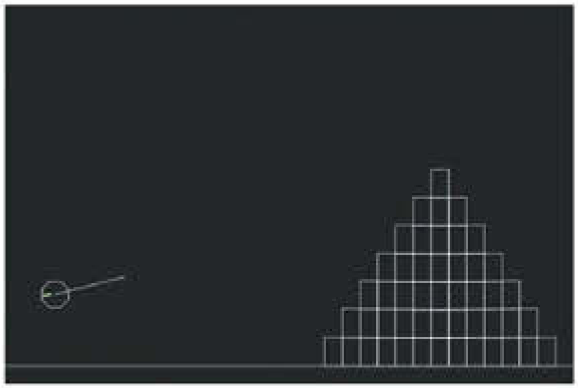
I've also passed an options object again. This time we flag the ground as `isStatic` since the ground doesn't normally move.

CREATING PHYSICS OBJECTS

Now let's create a rock. We'll be adventurous and use a hexagon, with radius 20px:

```
var rock = Bodies.polygon(170, 450, 8, 20);
```

To fling our rock, we need some elastic. We'll use a `Constraint` for this:



Simple demo The slingshot game we will be creating in this tutorial

```
var anchor = { x: 170, y: 450 },
elastic = Constraint.create({ pointA: anchor, bodyB: rock,
stiffness: 0.1 });
```

As you can see, one end of the Constraint will be fixed to pointA, an anchor at a fixed position in the scene. The other end is fixed to the centre of the rock we just created. I've set the stiffness pretty low to make the Constraint act more like elastic, to give us some firepower for our slingshot.

SETTING UP A TARGET

Next we need a target to smash. The engine supports Composite bodies, which are collections of Body, Constraint and even other child Composite objects that can be manipulated as a group.

We want a pyramid of blocks, which normally takes a bit of code to set up. Since this is a common requirement, Matter.js includes a Composites factory that can create stacks and pyramids easily, like so:

```
var pyramid = Composites.pyramid(450, 300, 13, 10, 0, 0,
function(x, y, column, row) {
    return Bodies.rectangle(x, y, 25, 40);
});
```

Matter.js supports Composite bodies: collections of Body and Constraint objects

I won't go through all the arguments here, but you can find them in the API docs via the project page. (Essentially, the pyramid factory uses a callback in which you create your bodies and the factory will stack them correctly for you.)

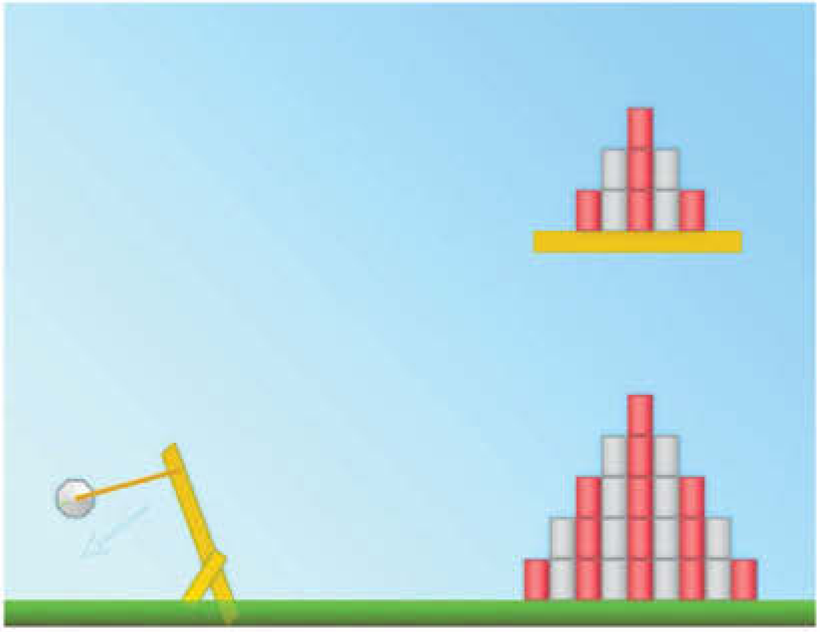
We now have all the bodies and constraints we need, so let's add them all to the world:

```
World.add(engine.world, [mouse, ground, pyramid, rock,
elastic]);
```

SETTING UP GAME LOGIC

The final part is to add a bit of game logic. We need the rock object to disconnect from the elastic when the player has let go of the slingshot, then reload with another rock.

For this, we'll bind an event to the engine. The tick event is called on every update of the engine. This normally means there are 60 ticks a second (at 60fps):



Graphical polish The Matter.js slingshot game we have created during this tutorial, after a little extra work

```
Events.on(engine, 'tick', function(event) {
    if (engine.input.mouse.button === -1 && rock.position.x >
    190) {
        rock = Bodies.polygon(170, 450, 7, 20);
        World.add(engine.world, rock);
        elastic.bodyB = rock;
    }
});
```

The code above does several things. First, it checks that no mouse button is currently being pressed; and given that, it checks that the rock has moved horizontally past its initial resting position while on the elastic.

If this is true, the rock should be released. The code creates a new rock body and attaches it to the elastic constraint. This means that the previous rock is no longer attached and is released, hopefully now flying towards our pyramid of blocks. What's more, we now have a brand new rock on the slingshot that is good to go.

Now we have everything set up correctly, the last thing to do is kick off the simulation:

```
Engine.run(engine);
```

Again, make sure you have wrapped all of the above code in a function and bound it to the window load event, as discussed at the start of the tutorial.

Hopefully when you open your page in the browser, the game should be ready to try out. If it doesn't work, check the console, then check the link to the tutorial code and compare the two. ◘

RESOURCE

SOURCE CODE

Want to go further, or contribute to Matter.js yourself? Find the code on GitHub at: *netm.ag/github-255*

Download the files here!

Find the GitHub repo for this tutorial at: *netm.ag/phasergit-260*

ABOUT THE AUTHOR
ANTON MILLS

w: *holler.com.au*

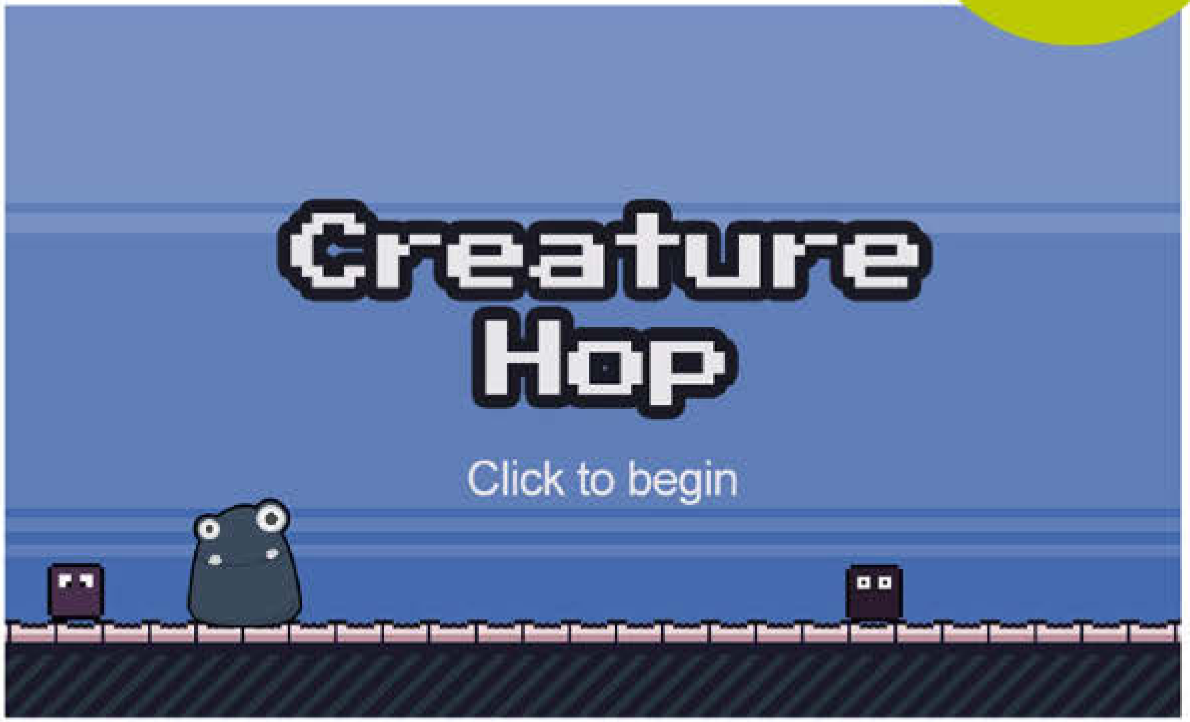
t: @antonmills

job: Technical director, Holler

areas of expertise: JavaScript, creative technology, game development

q: what was your childhood nickname?

a: Otter, because one kid said I had a face like an otter when I was in primary school. Kids can be so mean (sniff, sniff)



PHASER

DEVELOP A BROWSER GAME WITH PHASER

Follow **Anton Mills** as he guides you through the fundamentals of browser-based game development with the Phaser framework

GAME

The Wizard is a retro styled rogue-like game mimicking the golden 16-bit era, made with Phaser. This fun game is out now, and you can find it here: *netm.ag/wizard-260*

Now is a hugely exciting time for developers creating games for internet browsers, and over the past 12 months Phaser (*phaser.io*) has gone from strength to strength. This game framework provides a friendly and intuitive ecosystem for game development and also leverages the excellent Pixi.js (*netm.ag/pixi-260*) for outstanding Canvas and WebGL-rendering performance.

GETTING STARTED

You'll need Yeoman (*yeoman.io*) set up on your machine, so if you don't have it (and why the heck not?) follow the instructions on its website. With Yeoman up an running, start by installing a Yeoman project template for game development – this will save a lot of repetition when creating games.

To install the Yeoman project template generator, open a new terminal window and type the following (remember to prefix with `sudo` if you're on Mac):

```
npm install –g yo generator-phaser-official
```

The first thing we need to do is to create and compile a project to test everything. Create a new folder for your game and `cd` to it in your current terminal directory, then type:

```
yo phaser-official
```

Yeoman will ask you for the answers to few questions while creating the template: the game name (we've used 'netmag-phaser'), the version of Phaser you want to use (2.0.0), and your game's width (900) and height (480). When Yeoman has finished creating the project, check everything has been set up correctly by running `grunt` from the command line.

Your browser will open and the default game this project contains will start running. Have a play!

STRUCTURE AND STATES

Looking into the project's structure, there are a few key folders we will be working with, including:

- `/assets` – the folder containing the image assets
- `/game` – the game's JavaScript files
- `/game/state` – the game's states
- `/dist` – the compiled game for hosting online

You will spend the majority of your time inside the `game` and `states` directories; these are the core project files. Each game is made up of several 'states' – you can think of these as different stages of a game. In our game we will use just four states: Preload, Menu, Play and Gameover. You can see these JavaScript files in the `game/states` folder.

Phaser provides a friendly and intuitive ecosystem for game development

Each state is made up of a common layout. If you look at the template states you will notice the most common methods are Preload, Create and Update:

- **Preload** – offers a place to situate an asset or data preloading before the state is created
- **Create** – is where we instantiate objects, place graphics and set the state so it is ready to play
- **Update** – can be thought of as the state's main tick. This method is invoked at 60 times a second and contains the core of our game's logic

Now is a good time to download or clone the contents of this tutorial's GitHub repository (*netm.ag/phasergit-260*). Inside, there are a number of folders for different stages in the game, including one entitled `Assets`, which contains the imagery for this game. Start by copying the entire contents of this folder to the `/assets` folder in your game.

IDENTIFIERS

Jumping into the Preload.js state lets you set each of the copied assets to preload before our game starts.

▶

FOCUS ON

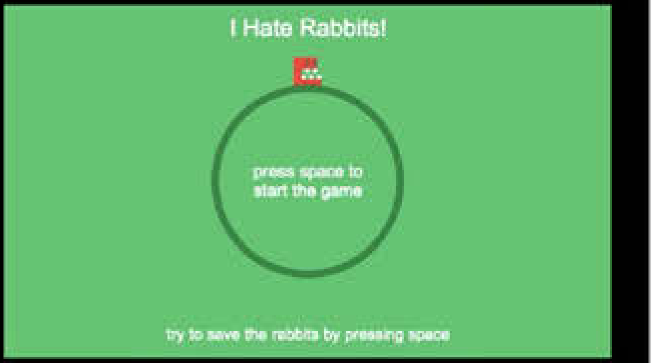
12 GAMES, 12 WEEKS

Thomas Palef is an engineer based in Paris. He had very little experience of game development but had always loved games, so after an initial week of research he decided to use the Phaser framework to jump in and learn how to develop games for himself. He set himself a challenge: develop 12 games in 12 weeks using Phaser. You can check out some of the games he made during this challenge on his website (*lessmilk.com/12games.php*).

Keeping the graphics minimal allowed Palef to focus on the game development and game logic itself, and not let the aesthetics take up his time. I think you'll agree that this lends his games a stylised, retro charm which only boosts their brilliance.

Take a look through his games and use them as inspiration for your own creations. Ask questions like: what ideas can you borrow? How can you make it unique? What can you change? There's a whole world of game development at your fingertips and browser-based games are extremely accessible, as you'll see by browsing through Palef's gallery of games.

The Phaser framework is built on top of Pixi.js, a dedicated rendering system for browsers. The frameworks are a fantastic match, coming together to help users create great games, quickly. *Read **net**'s interview with Thomas Palef at netm.ag/side-260.*

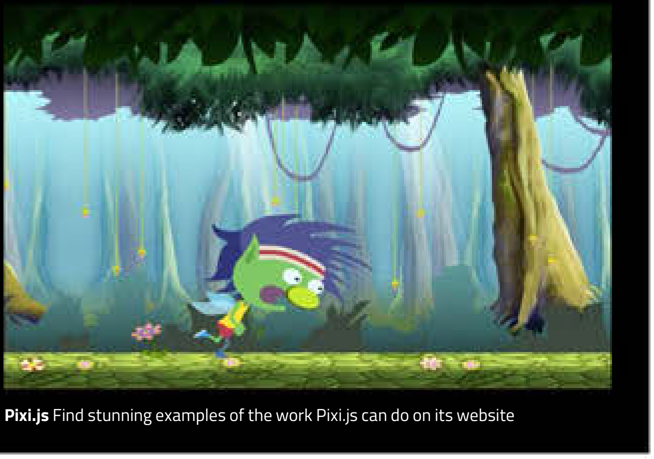


I Hate Rabbits!

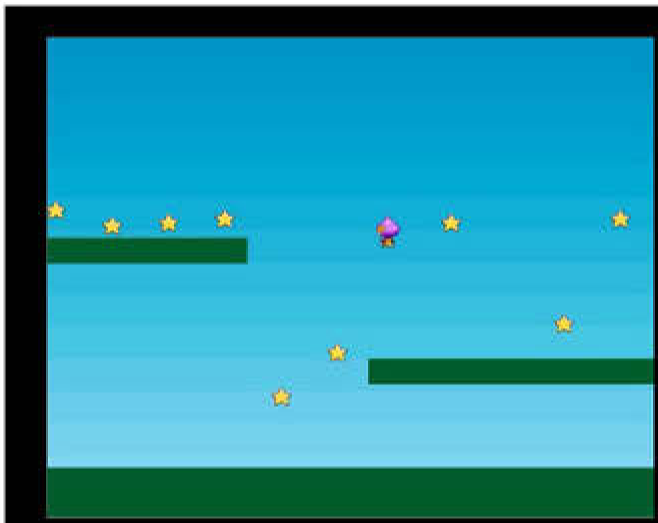
press space to start the game

try to save the rabbits by pressing space

Save the rabbits Explore Thomas Palef's creative and unusual game style



Pixi.js Find stunning examples of the work Pixi.js can do on its website



IN-DEPTH

GETTING STARTED

With Phaser being an extremely popular framework for game development, there's no shortage of learning materials out there. Your first stop for anything Phaser-related should be the excellent Phaser website (*phaser.io*), which boasts a huge range of examples, hosted with sample code, to walk you through each piece of Phaser's functionality. You can find the examples at *phaser.io/examples*. Of particular note is the Activity Feed at the bottom. This is a curated list of all things Phaser – very handy!

PhotonStorm (aka Rich Davey, the creator and lead developer of the Phaser framework) hosts a tutorial on creating your own Phaser-based platform game. This has been updated to the Phaser 2.0.0 release we use in this article and is another great learning resource. I would highly recommend jumping into this tutorial to cover some of the basics for platform-based movement with the framework's in-built physics system. You can find the tutorial online at *netm.ag/firstgame-260*.

Last but not least there is a Phaser ebook written by none other than Thomas Palef, of '12 weeks, 12 games' challenge fame, and it's fantastic! Check out the book's accompanying web page at *discoverphaser.com* for a little more information. It features some great examples of games that you will build throughout the chapters, it's up to date with the Phaser 2.0.0 release and will extend a lot of the functionalities covered in this article.

▶ Add the following to the `preload.js preload()` method:

```
this.load.image('menu', 'assets/menu.jpg');
this.load.image('background', 'assets/background.jpg');
this.load.image('player', 'assets/player.png');
this.load.image('creature_1', 'assets/creature_1.png');
this.load.image('creature_2', 'assets/creature_2.png');
this.load.image('creature_3', 'assets/creature_3.png');
this.load.image('gameover', 'assets/gameover.jpg');
```

The `load.image` method takes two parameters. The first is a string-based identifier we will use to reference the image if we need to create instances of it once it's loaded, and the second parameter is the path to the image.

Next up is the game's main menu scene. Open up the Menu.js state, remove the entire contents of the `create()` method and replace it with the following:

```
this.title = this.game.add.sprite(0, 0, 'menu');
```

Remember those identifiers we gave our assets at Preload? We simply have to use `game.add.sprite()`, give it a horizontal and vertical position of 0 and then use the `menu` identifier we specified in the Preload state for the main menu image.

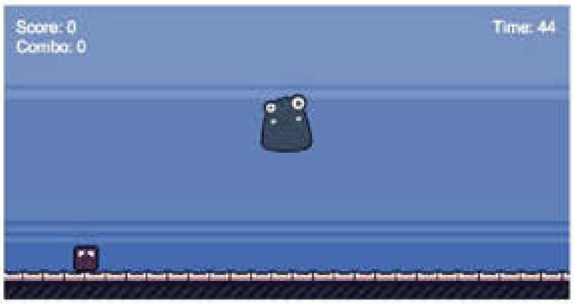
Test your progress by running Grunt from terminal to see the results load up in the browser. Your game should boot up to a new menu with graphics.

MECHANICS

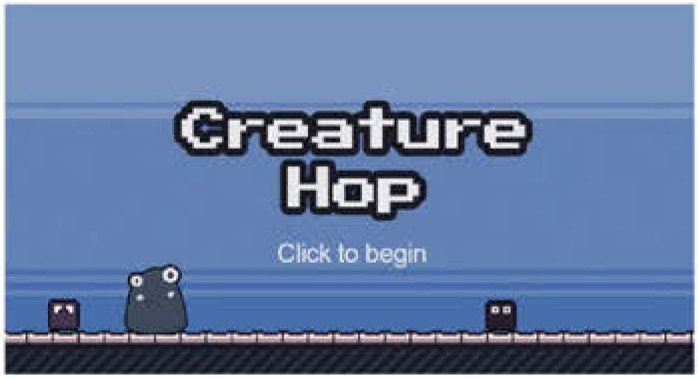
In this section we will look at the game mechanics, but to save lots of typing I've created the basic game class. Simply copy the Play.js from the folder entitled `3-Play` in the accompanying source code and overwrite your current Play.js in `/game/states`.

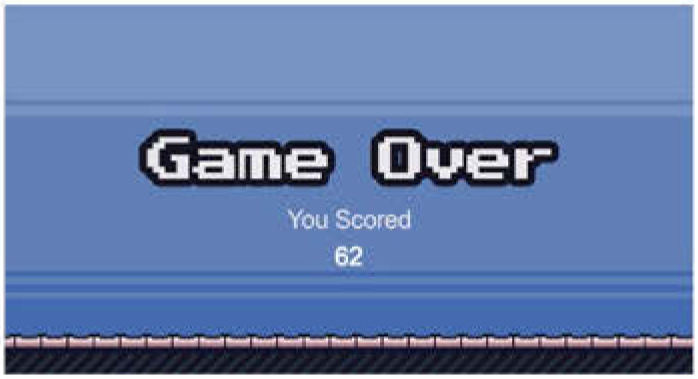
Inspecting the code, there are a few key lines of the framework to point out. Lines 10, 13 and 14 create a rectangle that is used as a boundary to keep objects within it. These lines also enable Phaser's built-in Arcade physics system.

Lines 17 to 21 create our player. We load the image the same way as before, but this time on line 20 we



Main state Here is the game's main state. The aim is to squash as many creatures as possible





Left At this point, our game's Menu state features new graphics and only a few lines of code

Right The final game state is the Gameover state. We have switched out the graphics and added a score

set the anchor. This essentially means the object's x and y position can be offset – so we position the player by its centre point (horizontally) and not its top left point.

We add the player to the physics system by using `game.physics.arcade.enable()` and setting the player's body to collide with the boundary. At this stage the player will react to gravity, and collide with the floor.

Lines 24 to 32 create an array of enemies using the same method as above. There are a few extra properties we create on each enemy that are used in their jumping functionality. This sets an intensity for their jump height and a delay between each jump. The logic for jumping is in the `Statesupdate()` method.

> These lines create a boundary rectangle and enable Phaser's Arcade physics system

Lines 39 through to 54 create the variables for score and time. They also use another of Phaser's functionality for rendering text. Using `game.add.text()` you can create a string of text on-screen while specifying fonts, fills and alignment. Phaser has great text-based functionality – we're only scratching the surface in this tutorial.

UPDATE METHOD

Now we can move on to the `update()` method, which is where the main game's logic resides. Lines 59 and 60 create shortcuts to the framework's input system. The `cursors` variable allows us to check if keys are down, using `cursors.right.isDown()`, for example.

We use this in lines 66 to 73 by listening to which keys are down, and if the player is not colliding with the boundary object we previously looked at. We can do this by using `player.body.onFloor()`. If this returns `false` then the player is in the air, and we give the player a velocity change using `player.body.velocity.x`. We don't have to use actual x-y values, we just give a velocity to x-y and leave Phaser's physics system to handle the rest. Our player can now move around while jumping.

At line 83 there is an example of how we update a Phaser text field. We simply use `setText()` and the text content we want to render. It will use the existing text style properties we instantiated it with.

An example of how to test for collisions between two sprites (not a physics-based collision) is to use the `Phaser.Rectangle.intersects method()`, as per line 113. We pass it two `sprites.bounds()` and it will return a `true` or `false`, based on if they are colliding or not. If the collision returns `false`, it hides the enemy until the enemy timer creates a new one.

GAME OVER

The rest is mostly just standard JavaScript, but there is one final part of Phaser to highlight – the `game.state.start()` method on line 129. Once this is invoked it ends the current state and moves to any state we specify as a string – in this situation, when the timer hits 0 we jump to the Gameover state.

In the Gameover.js, clear the contents of the `create()` method. Similarly to the Menu state, we will create a background image, but this time adding a dynamic text field to show the user's score.

Enter the following in the `create()` method:

```
this.gameoverbg = this.game.add.sprite(0, 0, 'gameover');
this.score_text = this.game.add.text(this.game.world.centerX,
325, this.game.score, { font: '32px Arial', fill: '#ffffff', align:
'center'});
this.score_text.anchor.setTo(0.5, 0.5);
```

Now save the Gameover.js file and run Grunt again. This time you should get the full menu screen, the game and a much improved Gameover screen.

Hopefully this has given you a taster for game development with Phaser. It's a great framework that will help you create some fantastic games.

GAME

A Ghosts and Goblins remake (woohoo!) was released using Phaser by Karza Games and wow is it impressive! Take a peek, you will love it: *int33h.com/test/gng*

Download the files here!

All the files you need for this tutorial can be found at *netm.ag/p5-261*

ABOUT THE AUTHOR

SCOTT GARNER

w: *scott.j38.net*

t: @scottgarner

areas of expertise: Creative technology, interactive design

q: who would play you in the movie of your life?
a: A computer generated viscacha

JAVASCRIPT

EXPLORE CREATIVE CODE WITH P5.JS

p5.js seeks to bring the power and flexibility of Processing to the web. **Scott Garner** introduces the new tool for creative coders

VIDEO

Watch an exclusive screencast of this tutorial created by Scott Garner at *netm.ag/p5Vid-261*

The new p5.js (*p5js.org*) is a library designed to bring the power of Processing (*processing.org*) to the web. It aims to introduce artists, designers and educators to the world of programming while also offering versatile tools to bring devs and engineers into the visual arts.

Let's dive in and create our first 'sketch'. Our goal is to build a drawing tool that transforms a simple image into a field of animated stars. First, we'll define a few global variables and write our `setup()` function. p5's `setup()` is run once, when the sketch is loaded, so it's the ideal place to handle initialisation.

```
var hintImage, skyImage, stars = [];
function setup() { ... }
```

Inside our `setup` function, we'll create a canvas and hide the mouse cursor so we can draw our own. By default, p5 adds an outline around shapes – we want to disable strokes in this case.

```
createCanvas(800,500);
noCursor();
noStroke();
```

Next, we'll load a pair of images. One will serve as the background – in this case, a night sky scene. The other will be the 'hint' image – the black and white design (seen overleaf) our final design will be based on. The idea is to put most of the stars over black pixels in our hint image, to recreate the design in our background scene. It would be easy to create these images with p5's text and drawing tools, but for the sake of brevity we'll use static assets.

```
hintImage = loadImage("//bit.ly/hintImage");
skyImage = loadImage("//bit.ly/skyImage");
```

THE DRAW FUNCTION

That's it for `setup()`! Another key function is `draw()`. It's called in a continuous loop, which is helpful for animations and adding elements over time.

```
function draw() { ... }
```

Within the Draw function, our first task is to fill the canvas with our background image. p5 doesn't automatically clear the canvas between `draw()` calls, so we need to do this every frame or we'll end up with some strange accumulation effects. To place a loaded image on the canvas, use the `image()` function and give x and y coordinates for positioning.

```
image(skyImage, 0, 0);
```

p5.js aims to offer versatile tools to bring devs and engineers into the visual arts

Next, we'll grab the current mouse location and store it as a p5.Vector using `createVector()`. This object comes with handy functions to deal with points in space, but we're mostly just using it as a container.

```
var position = createVector(mouseX, mouseY);
```

Using our newly-stored mouse position, we can draw our cursor. We'll set the drawing colour with `fill()` by passing RGB values and use `ellipse()` to draw a circle at the mouse location.

```
fill(255, 192, 0);
ellipse(position.x, position.y, 8, 8);
```

We only want to draw new stars while the mouse is pressed, so we'll check p5's `mouseIsPressed` before proceeding. If the mouse is down, we need to calculate a good place for our next star to end up. We'll do that with a custom function called `findPixel()`, which we'll define later.

Once we have a target, we'll create a new instance of a custom Star object (more below) and push it

▶

FOCUS ON

A BIT OF BACKGROUND

Processing is a programming language and development environment originally designed to teach the basics of computer programming within a visual arts context. Since its creation in 2001 by Ben Fry and Casey Reas, it has continued to expand and evolve into one of the most powerful platforms for creative coding. While screen-based drawing remains a central focus, it is now used for everything from generative sound projects to digital fabrication.

The goal of p5.js is to take the core ideas of Processing and adapt them to the modern web. This not only means a shift from Java to JavaScript, but also the introduction of new features like HTML DOM manipulation, and a broader reach than was previously possible. It also means you have the full power of the web at your disposal, allowing you to interface with other JavaScript libraries, access popular web-based APIs and easily target any device with a browser – from desktops to laptops to mobile phones.

One of the biggest advantages of working with p5.js is 13 years' worth of tutorials, examples and other learning materials created for Processing. In most cases, core principles translate directly, and code can often be adapted with only a few minor tweaks. Again, teaching is an essential element in p5, both for artists and designers interested in programming, and for programmers interested in the visual arts – so it's a great place to start, regardless of your skill level.

The creation of p5 is a collaborative development process led by Lauren McCarthy, with contributors from around the world. The project is always looking for enthusiastic developers, designers, artists, teachers, authors and, really, any other role you could imagine. If you'd like to get involved, please get in touch at hello@p5js.org. All contributions are welcome.

Announcing p5 p5.js was released to the public last summer. See the interactive announcement video at *hello.p5js.org*

▶ onto the end of our star array. If we end up with over 2,000 stars, we'll start discarding the oldest ones.

```
if (mouseIsPressed) {
  var target = findPixel();
  var star = new Star(position, target);

  stars.push(star);
  if (stars.length > 2000) stars.shift();
}
```

Finally, we'll loop through our array of stars and call `update()` and `draw()` on each of them. We'll dive into these methods later on.

```
for (var i = 0; i < stars.length; i++) {
  stars[i].update();
  stars[i].draw();
}
```

HELPER FUNCTIONS

Now `setup()` and `draw()` are in place we'll work on the helper functions and objects. First, we'll define a function that finds a resting position for each new star. All we need to do is check some random pixels in our hint image, using `get()` to see if they're black or white.

We actually only have to look at their red value, because in both cases the RGB values match. If we find a non-white pixel, we'll return it because we've hit our design. If not, we'll just return the last pixel checked and it will serve as a random star.

```
function findPixel() {
  var x, y;
  for (var i = 0; i < 15; i++) {
    x = floor(random(hintImage.width));
    y = floor(random(hintImage.height));

    if (red(hintImage.get(x,y)) < 255) break;
  }
  return createVector(x,y);
}
```

Last up is our custom Star object. Put simply, we want a reusable container that stores information about each star, and also provides a means for updating and drawing them. JavaScript doesn't provide a way to create classes in a traditional sense, but we can get the same result by defining a function and extending it for our own needs.

Each star has a few properties (starting position, final location and a randomly generated diameter), which we'll define in a 'constructor function' that is called for every new star created in our draw loop.

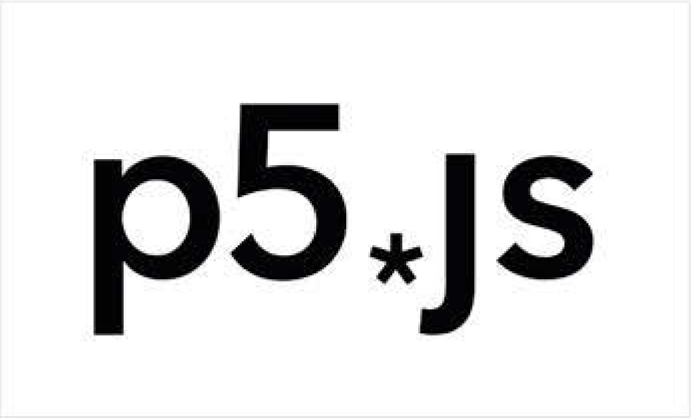
To give the stars a twinkle, the alpha value is determined using p5's noise() function

```
function Star(position, target) {
  this.position = position;
  this.target = target;
  this.diameter = random(1, 5);
}
```

Next we'll add an `update()` method to Star, which will use p5.Vector's `lerp()` to calculate a new location between a star's current and target positions. In this case, we're moving four per cent of the remaining distance for every draw, which effectively gives us an ease-out effect.

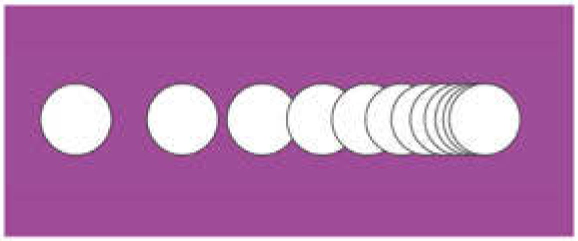
If we wanted to get fancy, we could also simulate some physics here, but for now we'll keep it simple.

```
Star.prototype.update = function() {
  this.position = p5.Vector.lerp(
    this.position,
```



Below left The background image – this night sky scene provides the setting for our animation

Below right The hint image, which dictates the positioning of our stars



Interpolation Each frame, linear interpolation gives us a new point along the path between the current star position and its destination

```
    this.target,
    0.04
  );
};
```

DRAWING STARS

Finally, Star's `draw()` method actually draws the star to the canvas. Once again, we're using `fill()` and `ellipse()`, although this time we're calling `fill()` with a grayscale value and an alpha value for transparency.

To give the stars a twinkle, the alpha value is determined using p5's `noise()` function. This returns the Perlin noise value for the specified coordinates, meaning you get a smooth, fluid sequence of random numbers. For the third parameter, we're passing a time-based value rather than a spatial value, so that the noise will animate over time.

```
Star.prototype.draw = function() {
  var alpha = noise(
    this.target.x,
    this.target.y,
    millis()/1000.0
  );

  fill(255, alpha * 255);

  ellipse(
    this.position.x, this.position.y,
    this.diameter, this.diameter
  );
};
```

That's it for our first sketch! Click and drag to see new stars appear and conform to the hint image.

WHAT'S NEXT?

From here, you might add some HTML elements to control variables using the p5.dom add-on or even add sound to the visuals using `p5.sound`.

We've only scratched the surface of what's possible with p5.js, and with new features and an official editor on the way, there's even more to look forward to. Have fun!

IN-DEPTH

POINTS ON A PLANE

Like many graphics applications, drawing in p5 is based on a coordinates system. Horizontal movement is determined by changes on the x axis, and vertical movement by the y. All of p5's basic drawing functions, such as `ellipse()` and `rect()`, use this system, as do advanced operations like grabbing an individual pixel's colour with `get()`. So, to draw a 100px square, 50px from the left edge of the canvas and 25px from the top, we'd use:

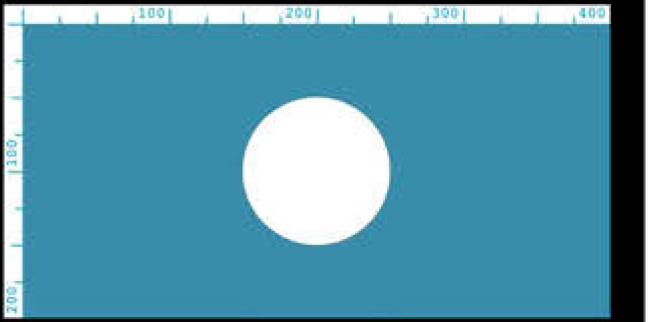
```
rect(50, 25, 100, 100);
```

Additionally, p5 offers some global variables helpful for positioning, like the width and height of the canvas. To draw a circle with a 100px diameter in the exact centre of the canvas, we'd simply divide the width and height in half.

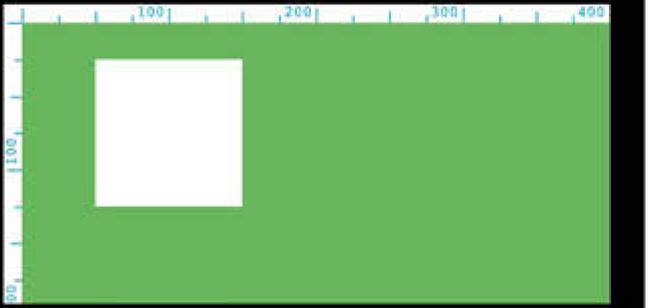
```
ellipse(width/2, height/2, 100, 100);
```

To begin animating these shapes, we'll need to pass variable parameters instead of constants. We can, of course, define our own variables, or use values provided automatically by p5. In the following code, for example, an ellipse is drawn wherever your mouse pointer happens to be on the canvas.

```
ellipse(mouseX, mouseY, 100, 100);
```



Ellipse ellipse() also takes x and y coordinates along with values for dimensions



Rectangle The first two arguments to rect() define the x and y position, the second pair determine the width and height

MacFormat presents
iPhone 6
Your companion manual
Mac FORMAT
FOR ALL YOUR APPLE NEEDS – for 21 years –
iClou
dem
What it is an
Image tricks for Pages
Unlock the power of Pages' photo editing
Power up OS X's Finder
Add new features to get more done, faster!
Use Windows hard disks
How to read and write to PC-formatted drives
116 pages of help and advice
We'll help you get the most from your Mac and iOS devices!
Kit rated
Messages
Calendar
Weather
Clock
Notes
Newsstand
Pro User Series
Do more with your iPad
Unleash your inner photographer, artist and more!
164 pages of photo, art, design and craft tips
94 Inspiring apps that will improve your creative side

HEAD TO HEAD

THREE.JS VS BABYLON.JS

Carl Bateman compares the pros and cons of the two competing WebGL frameworks for 3D programming

ABOUT THE AUTHOR

CARL BATEMAN

w: *WebGLWorkshop.com*

t: @CarlBateman

job: Senior software engineer

areas of expertise: C++, C#, MySQL, JavaScript, WebGL, OpenGL, GLSL

q: what's the dumbest thing you did in 2014?
a: Played a piano with a chainsaw, and went swimming in the Colorado River in March (5°C) – brisk!

NB: one of these is not true

	THREE.JS	BABYLON.JS
	three.js (*threejs.org*) is a general-purpose 3D rendering library with a focus on WebGL. It was developed by Mr Doob (Ricardo Cabello) with copious input from AlteredQualia, and is now an open source project hosted on GitHub.	Babylon.js (*babylonjs.com*) is a game-focused WebGL rendering API, developed as an open source project by Microsoft engineers David Catuhe, David Rousset and Pierre Lagarde. Hosted on GitHub, the download includes JavaScript and TypeScript versions.
DOCUMENTATION	three.js provides documentation – although sometimes incomplete, especially in the case of newly added or recently updated features – along with examples and an editor.	On the Babylon.js site you'll find documentation, examples, a sandbox, an editor, a live playground and online tutorial courses. However, sometimes it can be difficult to inspect the code.
COMMUNITY	There's a very active community with many videos, sandboxes, tutorials, around 10 books and at least one online course (Udacity's frankly excellent 'Interactive 3D Graphics' – *udacity.com/course/cs291*).	It's still early days for Babylon.js, but the number of tutorials is rapidly increasing, including some hosted by Microsoft Virtual Academy. There's also a community forum at *netm.ag/babylonforum-263.*
PHILOSOPHY	three.js is a general purpose rendering API. It can also render to `<canvas>`, `<svg>`, CSS 3D and DOM, providing a fallback if WebGL is unavailable.	Babylon is game-focused, using WebGL only with no fallback if it's unavailable. The team is ambitious and the scope and speed of development is impressive.
FEATURES	three.js.js provides incremental and binary loaders and plugins for Blender, Max, Maya and Revit. Keyframe and morph animation are supported. It has extensive light and shadow support – models can cast and receive shadow, and self-shadow. three.js supports post-processing effects (custom and supplied) and custom shaders. It has no built-in support for physics nor the esoteric but very useful CSG.	Babylon.js provides incremental and binary loaders and plugins for 3ds Max. Keyframe animation is supported. Light types include point, directional, spot and hemispheric, and models can cast and receive shadow but can't self-shadow. Babylon.js supports post processing effects (custom and supplied) and custom shaders. It comes with a physics engine, cannon.js, and has a CSG library based on csg.js.

VERDICT

There's no clear winner as both are exceedingly good at what they do. The differences between the two are mainly in syntax and usage. While three.js does have a broader feature set, Babylon.js is catching up fast. Babylon's focus on games means it can be quicker to get up and running with more features immediately available, while three.js' maturity means more third-party resources, and it provides numerous renderers, cameras and lighting.

FACT FILE

ALTERNATIVES

PlayCanvas is a professional, open source game engine (*netm.ag/play-263*) with free and paid hosting plans and a full Unity-style visual editor

scene.js a free, open-source 3D visualisation engine

processing.js is a JavaScript implementation of Processing which supports both canvas and WebGL, and is heavily focused on creative coding

EXCHANGE

Practical advice from industry experts

★TOP QUESTION

Why do you think that D3 has risen to its current popularity?

Jani Karhunen, Kuopio, FI

D3.js is a JavaScript library for manipulating documents based on data. **D3** helps you bring data to life using HTML, SVG and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

Download the latest version (3.4.6) here:

Open visualisations D3 is open and standards-based, automatically leveraging browsers' built-in features

SM1: You may be too young to remember this, but it wasn't always cool to support web standards. Fortunately, the browser wars are past their peak, and now everyone agrees that standards are good. D3 is entirely open and standards-based, so using D3 means you automatically leverage all the built-in features of every browser. As browsers get better, D3 gets better along with them. Plus, learning to use D3 is a great way to deepen your skills with JavaScript, HTML, CSS and SVG.

NODE.JS

EVENT LOOP

What is an event loop in Node.js?

Chandan Kumar, New Delhi, IN

SM3: The event loop lives from the point at which a script is executed until the point at which there are no more callbacks left to perform.

Take a HTTP request. When the response is received, a message containing the callback is added to a queue. The event loop polls that queue and when it finds a message executes the callback. Further message polling is then halted until all calls on the stack have returned.

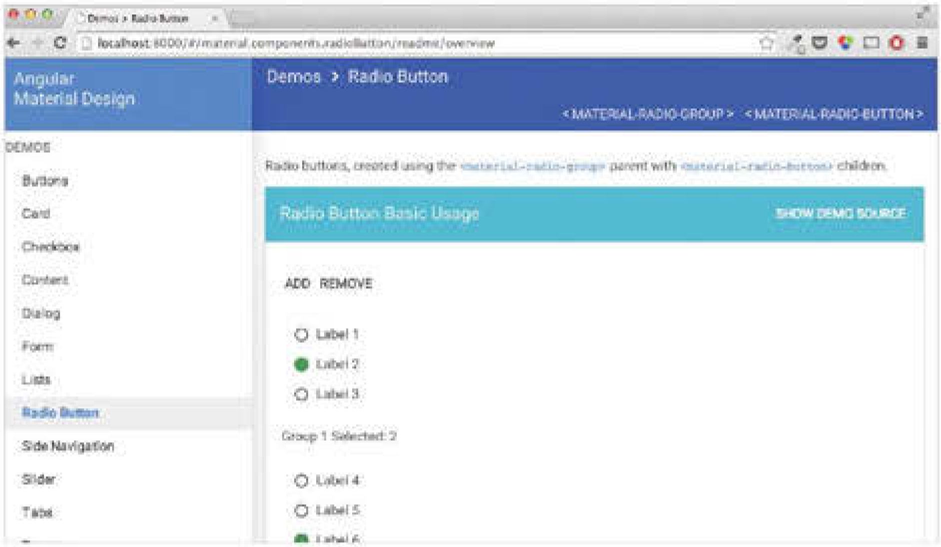
WEB TOOLS

NEXT GENERATION

What new and upcoming thing that affects web developers' lives are you most excited or frustrated about?

Brian Kardell, Vermont, US

RS: In-browser tools. That single answer actually answers all four parts of that question. Chrome (to me) is leading the way, adding amazingly useful and powerful tools to dev tools (like the recent canvas frame-by-frame inspection). Firefox's tools have been progressing really well in the last couple of years and there's even a rumour that IE11's dev tools are actually pretty damn good. The flip side is every other browser! Any time you have to support an Android browser (not Chrome) you're left with zero tools (which is when I turn to weinre and *jsconsole.com*). But then you're in no man's land, and forget about step debugging! The real hope I have here is the older mobile browsers will phase out (or certainly faster than IE6 did).



Radio buttons, created using the <material-radio-group> parent with <material-radio-button> children.

Material design Angular's Material Design module includes accessible element hints for the browser

ANGULARJS
GOVERNMENT GUIDELINES

What's the best way to start to make sure I'm meeting government accessibility guidelines? Any best practices?

Adrian Roselli, New York, US

NA: There are few Angular-specific considerations. Accessibility guru Marcy Sutton (@marcysutton) gave an excellent talk on Angular and accessibility that you can see at *netm.ag/access-262*. Here are some of the highlights:

- **Keyboard bindings: If your interfaces use custom click handlers, be sure to also enable ways to interact with those elements via the keyboard**
- **Native elements: When writing directives for custom UI elements, try to include the equivalent native element in your markup**
- **ARIA: While Angular does a lot for you, you still have to know your tools. The ngAria module provides Angular bindings for ARIA roles**

LOADING DEPENDENCIES
MANAGING LOCAL SCRIPTS

What's the best way to manage running local JS scripts when you're loading library dependencies (in this case, jQuery)?

Simon R Jones, Cambridge, UK

JD: Personally, when I am starting a new project, I tend to use CodeKit (*incident57.com/codekit*) to download the latest versions of libraries I need in one click. CodeKit also helps make sure you are running the latest versions of the libraries, and has a whole bunch of helpful tools you can use from the UI, but it's Mac-only. As far as managing and loading dependencies goes, I'm working on a couple of fairly large projects at the moment, using several complex custom scripts, for which I'm using RequireJS (*requirejs.org*).

WEB PERFORMANCE
OPTIMISING PERFORMANCE

How do you optimise performance? AJAX Req, screen repaints, lean JS?

Tanner Godarzi, Huntington Beach, US

PL: The first step in optimisation is to plan what your application is going to do and keep control of it. Only include the third-party JavaScript libraries that you really need and be aware of where your functionality will result in AJAX requests or screen repaints.

One way to do this would be to create abstractions around things like the AJAX request so that you know that all requests are made from a central place. Knowing this you can keep an eye on usage quite easily (for example, logging). You can also try to centralise functionality that touches the DOM and causes screen repaints; some UI binding libraries already do this. CSS can be harder to track down manually.

Even if you've tried to keep a manual eye on things you may still end up with

▶

FEATURING.....

SCOTT MURRAY

Scott (referred to as SM1) is a teacher and code artist specialising in data visualisations

w: *alignedleft.com*

TERO PARVIAINEN

Tero is an independent software developer and the author of *Build Your Own AngularJS*

w: *teropa.info*

NATE ABELE

Nate is the founder of Radify and a member of the AngularUI team. Tweet him @nateabele

w: *nateabele.com*

JON DUCKETT

Jon makes books on frontend design and development for London-based company Wagon

w: *javascriptbook.com*

REMY SHARP

Remy is a developer and owner of Left Logic, based in Brighton. He founded the Full Frontal conference

w: *remysharp.com*

TAMMY EVERTS

Web performance geek Tammy is a solution evangelist at Radware

w: *webperformance today.com*

PHIL LEGGETTER

Phil is a developer evangelist at Caplin Systems, where he's open sourcing BladeRunnerJS

w: *leggetter.co.uk*

STACEY MULCAHY

Stacey (referred to as SM2) is a Microsoft technology evangelist for Windows 8 and a 'tinkerer'

w: *thebitchwhocodes.com*

SIMON MCMANUS

Simon (referred to as SM3) is a freelance JavaScript engineer specialising in Node.js

w: *simonmcmanus.com*

a few performance issues that you can't track down. Or you may simply be convinced the problem is with CSS.

But don't worry: Browser Dev Tools to the rescue! Chrome, Firefox and even Internet Explorer have some fantastic tools for tracking down unwanted code bottle necks, network requests and animations that make your site act and feel 'janky'.

ANGULARJS

MODELLING

Angular doesn't really provide a 'model' layer – how should I model my data?

Stephen Fernandez, Holmston, UK

NA: Contrary to popular belief, Angular is not so much a framework as a library or toolkit. It provides some generic ways to organise your code, as well as some very powerful primitives to compose UIs and application logic, but how you compose them is up to you. Your 'data model' is `$scope`. Fortunately, you have a number of options. One is to box your objects in custom 'classes', like so:

```
function Widget(properties) {
 angular.extend(this, properties);
}

angular.extend(Widget.prototype, {
 custom: function() { /* ... */ }
});

myModule.controller("WidgetsController",
function($scope) {
angular.extend($scope, {
      widget: new Widget({ foo: true, bar:
false })
 })
});
```

The de-facto library for modeling data is probably Restangular (*netm.ag/restangular-262*). This provides a highly fluent, configurable way to bind complex client-side logic to data coming from API endpoints.

My company has published a similar, though much simpler library (*netm.ag/radify-262*), which provides conveniences for binding UI logic to API endpoints, but is centred around hypermedia. As such it makes some assumptions about your API to keep your frontend decoupled and flexible.

ANGULARJS

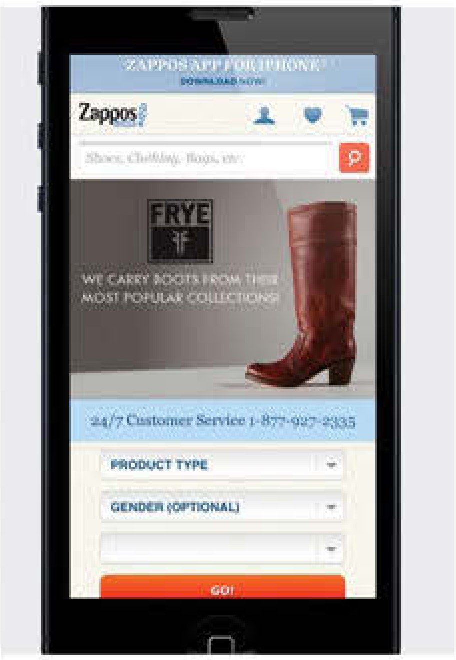
APP SEO

How can I maximise SEO in my Angular app?

Michael Wiley, Massachusetts, US

NA: As with any context in which search engines meet JavaScript, you're going to have to make some trade-offs. Often the dynamic parts of the apps I build live behind a login wall, and aren't expected to be indexed. For public-facing parts, I mark them up accordingly and keep dynamic content progressive.

One approach is to create Angular 'widgets': applications that occupy a small section of the page that they can progressively enhance by dynamically replacing its content (the BBC handles tabular data on heavily-trafficked pages like this). There are also services like Prerender.io (*prerender.io*) that you can use to crawl your apps in a way that takes JavaScript into account. Finally, Angular has announced that by the end of 2014, GoogleBot will take JavaScript fully into account when crawling applications, so this problem will be obviated entirely ... for one search engine, anyway.


ZAPPOS APP FOR IPHONE
DOWNLOAD NOW
Zappos
Shoes, Clothing, Bags, etc.
FRYE
WE CARRY BOOTS FROM THEIR MOST POPULAR COLLECTIONS
24/7 Customer Service 1-877-927-2335
PRODUCT TYPE
GENDER (OPTIONAL)
GO!

Mobile UI Keep it clear, simple and concise

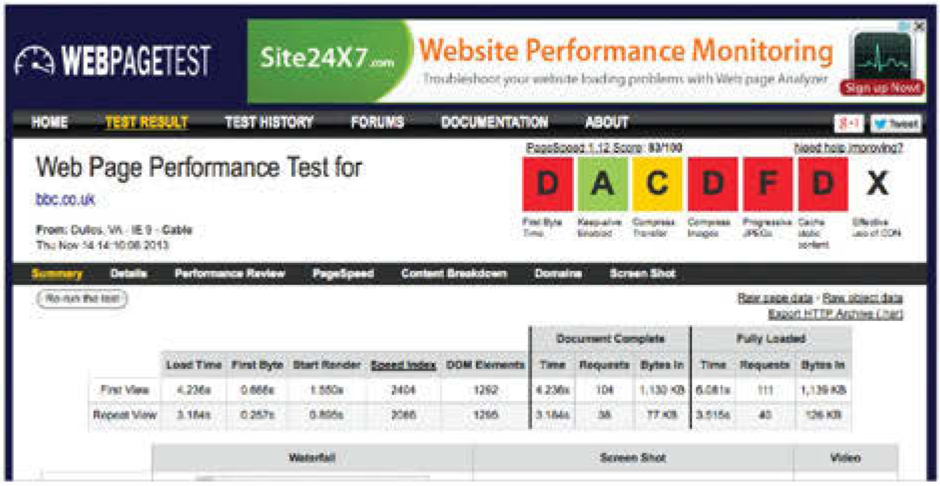
USER EXPERIENCE

MOBILE-FRIENDLY REDESIGN

I'm trying to redesign my site. What are some good tips to make my UX more mobile-friendly?

J. Edgar Montes, Michigan, US

TE: The best tips for designing a mobile-friendly site are actually universal for all platforms: keep it clear, simple and concise. Images are bandwidth hogs, so make sure they're optimised and compressed. Limit JavaScript. Don't make people scroll more than two screen lengths. Make calls to action prominent. Don't get sucked into bells and whistles. Be tough about cutting page elements that aren't necessary. And remember to test,



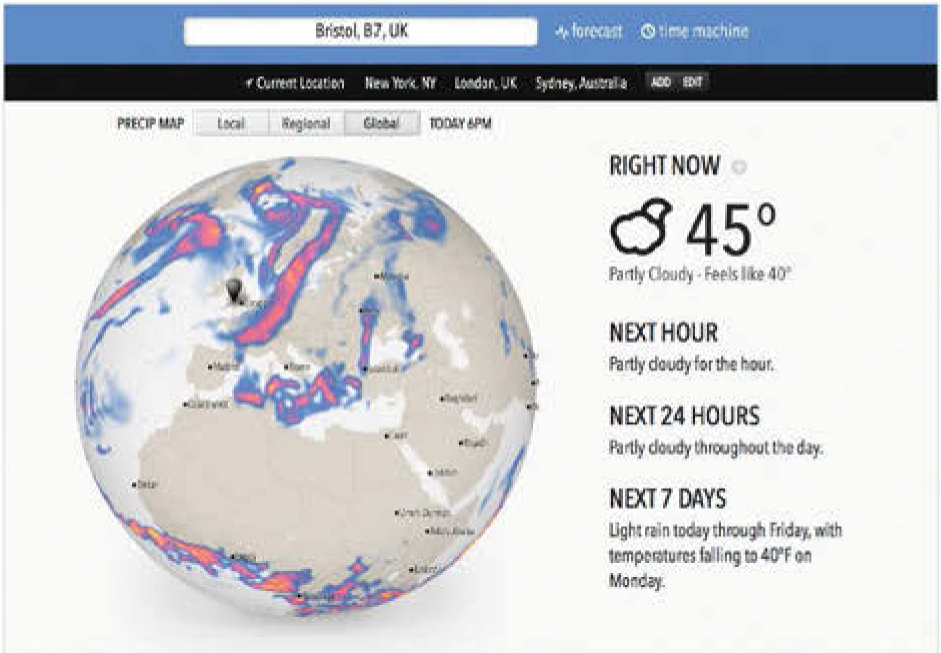
Web Page Performance Test for

bbc.co.uk

From: Dulles, VA - IE 9 - Cable
Thu Nov 14 14:10:08 2013

						Document Complete			Fully Loaded		
	Load Time	First Byte	Start Render	Speed Index	DOM Elements	Time	Requests	Bytes In	Time	Requests	Bytes In
First View	4.236s	0.668s	1.550s	2404	1262	4.236s	104	1,130 KB	6.081s	111	1,139 KB
Repeat View	3.184s	0.257s	0.805s	2086	1266	3.184s	38	77 KB	3.515s	40	126 KB

Page testing Use *webpagetest.org*. Look at your page's waterfall chart if you score less than A for keepalives



RIGHT NOW

45°

Partly Cloudy - Feels like 40°

NEXT HOUR

Partly cloudy for the hour.

NEXT 24 HOURS

Partly cloudy throughout the day.

NEXT 7 DAYS

Light rain today through Friday, with temperatures falling to 40°F on Monday.

Native apps *forecast.io* proves it is possible to create non-native apps with great performance

test, test – before and after launch – on a variety of devices, browsers and connection speeds.

HTML5 AND JS VS NATIVE

NATIVE APP EXPERIENCE

What's stopping HTML5 and JS apps performing as well as native apps?

Jérôme Lille, Strasbourg, FR

RS: People. The people who actually author those apps. Just look at *forecast.io* on an iOS device. Care and attention has gone into the effects, transitions and code so that it feels like a native experience (one that feels natural on the platform). The web platform is capable in most areas of application: news, utilities, lifestyle. But the devil is in the detail and, sadly, the truth is most designers and developers aren't working hard enough to achieve that experience parity to native apps.

WEB PERFORMANCE

STARTING OUT

Where's the best place to start [with web performance]? What can you do first to have the most impact?

Westley Knight, UK

TE: Start by ensuring you've enabled keepalives (keeping TCP connections open) and resource compression. Combined, these two techniques can improve start render time by up to 52 per cent and load time by up to 31 per cent.

Test your page using a tool like WebPageTest (*webpagetest.org*). If you score less than an A for keepalives, look at the waterfall chart for your page. If you see a lot of orange bars, you have a problem. Check you have the proper configuration on your servers and load balancer. The average web page is more than 1MB in size. Compressing resources can reduce the number of bytes sent over the network. Make sure you're following Google's best practices for enabling compression (*netm.ag/payload-249*).

INTERNET OF THINGS

DEFINING TERMS

How relevant is the term IoT?

Will McIntyre, Cincinnati, OH

SM1: It's a vague term. Is it outdated? Not entirely. The Internet of Things is about connected devices, connected things, through the internet. It doesn't really encompass at least one important aspect of connected devices, and that is data. Data, and the analysis of it, is where we get not just connected devices, but smart ones. Many people choose to call it the 'Sensornet'. I prefer Seb Lee-Delisle's 'ST4I' – Stuff that talks to the Interwebs.

▶

3 SIMPLE STEPS

What's the best way for someone who's not into hardware to get started?

Ricky Robinett, Brooklyn, NY

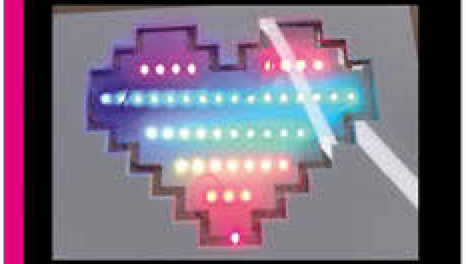
SM2: If you're new to hardware, give your first project the best start with some forward planning ...

KEEP IT SIMPLE

+ Best way to get started is to give yourself a small project with only a few moving parts. Keep your complex ideas for future projects.

CHECK THE PARTS

+ Order starter kits or inventor kits – and check you have everything you need. Missing parts can stall a project, and time away is time unmotivated.



DON'T TEST NEW GROUND (YET)

+ Perhaps choose a project that is a variation of something that has already been done, so you have a bit of guidance and something to aim for.

ANGULARJS
TITLE AND DESCRIPTION

How do you insert title and description into the head without adding a controller to the `<html>` tag?

Milton Jackson, Connecticut, US

TP: As long as the `<head>` section is inside the Angular app, which would mean having the `ng-app` attribute on the `<html>` element, you can attach custom directives to the `<title>` and `<meta name="description">` elements and insert content with those. You can also always use `angular.element` to do jQuery-like DOM operations. Using `angular.element` for anything other than local DOM access inside directives is not really the Angular way, but if there's no other way around the issue, then it is possible.

NODE.JS
SPLITTING UP

What do you think of the split between Node.js and IO.js (*netm.ag/schlueter-264*)?

Kate Simpson, Chicago, US

SM3: I'm always pleased to see efforts to be more open and encourage community participation. Node's active community is its greatest strength, and I'm hoping this can make it stronger rather than cause division.

I think Jeff Attwood (*blog.codinghorror.com/oh-yeah-fork-you*) says it well: "Forking is incredibly difficult to pull off. It is a painful, but necessary, part of the evolution of open source software." It's good to see that the IO.js developers have the intent to merge back with the original project "at some point in the future".

RECURSION
RECURSION EXPLAINED

Where's the best explanation of recursion in JavaScript for a designer with no programming background?

Duke Branding, Los Angeles, US

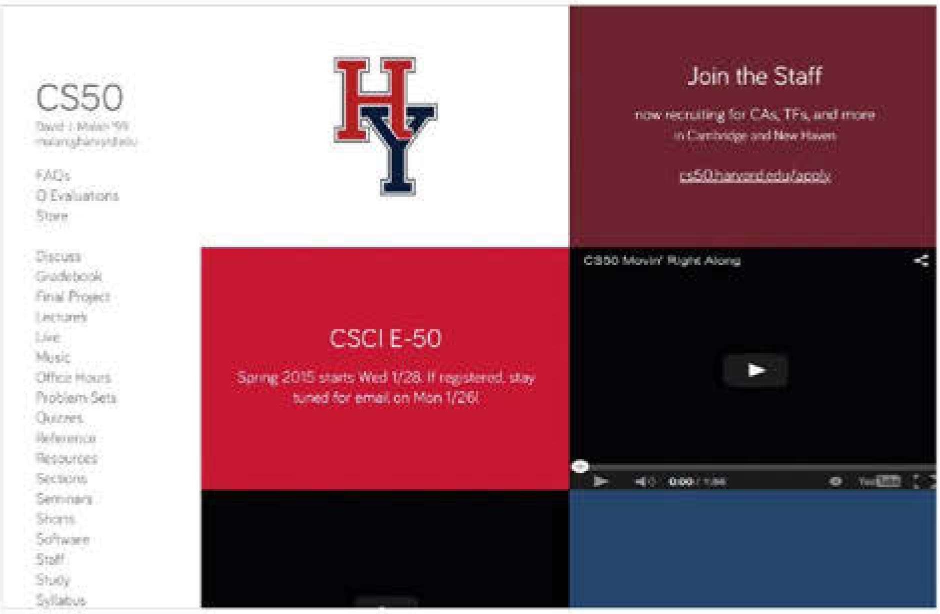
JD: Harvard University made lots of interesting videos for its Introduction to Computer Science course (*cs50.harvard.edu/shorts*). They aren't all focused on JavaScript, but they contain some helpful descriptions of computer science concepts for those that are new to programming. If you want to explore recursion, check out Week 4.

WEB PERFORMANCE
GOOGLE RANKINGS

Do we have evidence that performance has any bearing on our Google ranking? If so, what should we prioritise?

Matt Moore, UK

TE: The hard evidence swings both ways. I could point you to case studies demonstrating a connection between faster pages and improved organic search rankings, or I could point you to an SEOMoz survey showing a negative correlation between page response time and search ranking. At the end of the day, there are enough other proven benefits to improving performance – page views, bounce rate, conversions, revenue, user satisfaction – that the SEO question, while interesting, is somewhat moot.



Join the Staff

now recruiting for CAs, TFs, and more in Cambridge and New Haven

cs50.harvard.edu/apply

CSCI E-50

Spring 2015 starts Wed 1/28. If registered, stay tuned for email on Mon 1/26!

Recursion explained Harvard University has created informative videos for its Computer Science course

Ones and zeros io.js began as a fork of Node.js and aims to provide faster and predictable release cycles

TEXT EDITORS
FAVOURITE TOOL

What do you use for writing your JS?

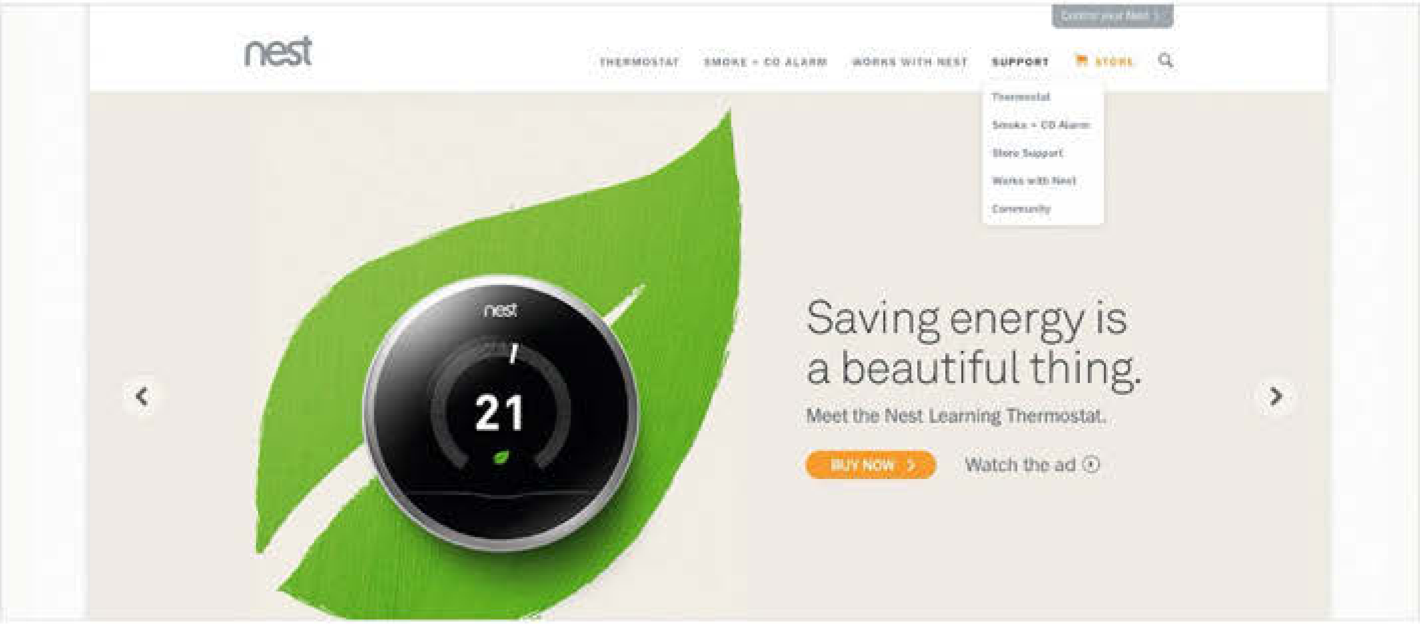
Sora Nana, JP

PL: I use Sublime Text. In the recent past I used TextMate and Cloud 9 IDE. In the distant past I've used Visual Studio, Eclipse and a number of editors that you may consider more enterprise development focused.

What I've come to believe is that you should choose an editor based on enabling a productive development workflow and a visual style you like. A customisable visual presentation is possible with most editors now; font, font-size, code formatting and colours can make you feel more at home with your editor.

When it comes to an editor a healthy plugin eco-system is key to getting your preferred workflow – or creating one. For example, if you want real-time feedback on your JavaScript syntax then choose an editor that supports something like JSHint (*jshint.com*). If you frequently use code snippets (common pieces of script), make sure that features is natively available or available as a plugin.

If the application you're working on has a large codebase then some form of dependency analysis and suggested auto-completion (*en.wikipedia.org/wiki/*



Nest value Nest has built up its considerable worth through its close access to customers

Autocomplete#In_source_code_editors) can be very handy because there's no way you're going to remember the entire codebase. There's lots more to consider, but it really depends on the workflow you're trying to achieve.

INTERNET OF THINGS

NEST VALUE

Are Nest's couple of sensors really worth $3.2 billion?

Sergi Golubev, London, UK

SM1: The value is not in the sensors, it's in the data and the access to consumers in the privacy of their homes. The hardware is just presentation. The amount of money Nest can make with that data and partnerships with utility companies will last as long as the hardware does. That is year after year of profits, with only one initial hardware cost.

ANGULARJS

DATA PERSISTENCE

What is the best data persistence library for AngularJS? Think socket updates, ORM, adapters, serialisers and so on.

Jon Koops, Hilversum, NL

TP: This is an area where AngularJS itself has very little to say at the moment, since it doesn't really have a model layer for which to do persistence. You can always roll your own, and for that a generic JS persistence solution like Hoodie, BreezeJS or Firebase will work just fine. In Angular 2.0 this will change – see *netm.ag/angular2-257.com* for more info.

COMMUNICATIONS

VIDEO TUTORIALS

Does doing web videos to teach coding help you with your own work quality or client explanations?

Tristan Bailey, UK

RS: Absolutely. When you're recording a video, you're editing your thought process for an audience, and it takes practice (and listening to feedback if you can). Clearly communicating an idea, either to yourself or to a client, is a core skill.

For example, if you encounter a problem, explaining it to the stuffed toy in the corner of the office will often yield a solution. Not because the toy is talking back, but because you've used a different part of your brain to break the problem into digestible morsels.

NODE.JS

HEROKU ALTERNATIVES

Could you recommend some alternatives to Heroku for Node.js hosting? I want to take a step past 'hello world', but I don't want to splash out over £100/year for something to hack on.

Clint Milner, Buckinghamshire, UK

SM3: If you're already using Heroku, it's worth exploring the options available as they will take you far beyond a hello world application. The list of add-ons (*addons.heroku.com*) gives a good idea of the kinds of external services that can be integrated. If you just need a personal hack space, you should be able to do most things without incurring any costs.

Or if Heroku really isn't for you, there is a well-maintained list of hosting providers available on the Joyent wiki: *netm.ag/joynet-264*.

ANGULARJS

URL BEST PRACTICE

What is the best way to configure URLs in one-page AngularJS apps so they don't break on page reload with html5mode?

Colm Morgan, Berkshire, UK

TP: For handling reloads with html5mode, some server configuration is needed. This is not just an Angular issue, but a general HTML5 History API issue. When you navigate from `/` to `/foo` in `html5mode`, Angular takes care of routing right in the browser. But when you reload `/foo`, you'll hit the server and get 404 because the browser won't know it's a special URL. You need to have your server respond with the same HTML for both `/` and `/foo`. For example, you can configure Apache or Nginx to rewrite `/foo` to `/`.

COLOPHON

@netmag /netmag +netmagazine *flickr.com/photos/netmag* netmag@futurenet.com *net.creativebloq.com*
Future PLC, Quay House, The Ambury, Bath, BA1 1UA +44 (0)1225 442244

EDITORIAL

Editor **Oliver Lindberg** *oliver.lindberg@futurenet.com*
Production editor **Ruth Hamilton** *ruth.hamilton@futurenet.com*
Art editor **Mike Brennan** *mike.brennan@futurenet.com*

EDITORIAL CONTRIBUTIONS

Nate Abele, John Allsopp, Justin Avery, Carl Bateman, Zack Bloom, Liam Brummitt, Geri Coady, Mark Dalgleish, Rob Dodson, Jon Duckett, Dan Edwards, Josh Emerson, Tammy Everts, Jonathan Fielding, Jack Franklin, Dan Frost, Scott Garner, Peter Gasston, Sebastian Golasch, Mat Groves, Aaron Gustafson, Benjamin Howarth, Tomomi Imura, Jimmy Jacobson, Mark Jones, Matt Keas, Steve Klabnik, Westley Knight, Vitaly Kondratiev, Gion Kunz, Patrick H Lauke, Phil Leggetter, Scott van Looy, Joe Maddalone, Kevin Mandeville, Tom Maslen, Eric Mann, Alex Matchneer, Mark McDonnell, Simon McManus, James Miller, Anton Mills, Stacey Mulcahy, Scott Murray, Dan Nagle, Den Odell, Kamil Ogórek, Tero Parviainen, Neil Renicker, Lukas Ruebbelke, Remy Sharp, Adam Simpson, Andi Smith, Karolina Szczur, James Williams, Nicholas Zakas

ART CONTRIBUTIONS

Martina Flor, Tobias Hall, Xavier Ho, Linzie Hunter

MANAGEMENT

Content and marketing director **Nial Ferguson** *nial.ferguson@futurenet.com*
Head of content & marketing, photography, creative and design **Matthew Pierce** *matthew.pierce@futurenet.com*
Group editor-in-chief **Dan Oliver** *dan.oliver@futurenet.com* Group art director **Rodney Dive** *rodney.dive@futurenet.com*

ADVERTISING Advertising sales director **Suzanne Smith** *suzanne.smith@futurenet.com*
CIRCULATION Trade marketing manager **Juliette Winyard** *juliette.winyard@futurenet.com*
PRODUCTION Production controller **Nola Cokely** *nola.cokely@futurenet.com*
Production manager **Mark Constance** *mark.constance@futurenet.com*
LICENSING International director **Regina Erak** *regina.erak@futurenet.com*
SUBSCRIPTIONS Phone our UK hotline 0844 848 2852; international +44 (0)1604 251 045
Subscribe to net online at *myfavouritemagazines.co.uk*

Future

Future is an award-winning international media group and leading digital business. We reach more than 49 million international consumers a month and create world-class content and advertising solutions for passionate consumers online, on tablet & smartphone and in print.

Future plc is a public company quoted on the London Stock Exchange (symbol: FUTR). **www.futureplc.com**

Chief executive Zillah Byng-Maddick
Non-executive chairman Peter Allen
Chief financial officer Richard Haley

Tel +44 (0)207 042 4000 (London)
Tel +44 (0)1225 442 244 (Bath)

abc

Print 10,632 **Digital** 3,672
The ABC combined print, digital and digital publications circulation for Jan–Dec 2013 is
14,304
A member of the Audited Bureau of Circulations

FSC
www.fsc.org
MIX
Paper from responsible sources
FSC® C007184

recycle
When you have finished with this magazine please recycle it.

We are committed to only using magazine paper which is derived from well managed, certified forestry and chlorine-free manufacture. Future Publishing and its paper suppliers have been independently certified in accordance with the rules of the FSC (Forest Stewardship Council).

THE JAVASCRIPT HANDBOOK

The ultimate guide to enhancing your sites with the most important programming language in the world

This all-new guide includes everything you need to do more with JavaScript, *the* programming language of the web. Through 27 practical projects, you'll learn how to speed up both your workflow and the performance of your sites and apps. You'll also discover the latest cutting-edge tools and techniques, how to build great apps with the best JavaScript frameworks, design 3D experiences with WebGL, create stunning user interfaces, and much more.

Become a JavaScript master today!